COPY

MORRISON & FOERSTER LLP
MICHAEL A. JACOBS (Bar No. 111664)
mjacobs@mofo.com
MARC DAVID PETERS (Bar No. 211725)
mdpeters@mofo.com
755 Page Mill Road
Palo Alto, CA 94304-1018
Telephone: (650) 813-5600
Facsimile (650) 494-0792

ADR

ORIGINAL FILED
AUG 12 2010
Richard W. Wieking
Clerk, U.S. District Court
Northern District of California
San Jose

E-filing

BOIES, SCHILLER & FLEXNER LLP
DAVID BOIES (*Pro Hac Vice* Pending)
dboies@bsfllp.com
333 Main Street
Armonk, NY 10504
Telephone: (914) 749-8200
Facsimile: (914) 749-8300
STEVEN C. HOLTZMAN (Bar No. 144177)
sholtzman@bsfllp.com
1999 Harrison St., Suite 900
Oakland, CA 94612
Telephone (510) 874-1000
Facsimile: (510) 874-1460

ORACLE CORPORATION
DORIAN DALEY (Bar No. 129049)
dorian.daley@oracle.com
DEBORAH K. MILLER (Bar No. 95527)
deborah.miller@oracle.com
MATTHEW M. SARBORARIA (Bar No. 211600)
matthew.sarboraria@oracle.com500 Oracle Parkway
Redwood City, CA 94065
Telephone: (650) 506-5200
Facsimile: (650) 506-7114

Attorneys for Plaintiff
ORACLE AMERICA, INC.

UNITED STATES DISTRICT COURT

NORTHERN DISTRICT OF CALIFORNIA

CV 10-03561 LB

ORACLE AMERICA, INC.

Plaintiff,

v.

GOOGLE, INC.

Defendant.

Case No.

COMPLAINT FOR PATENT AND COPYRIGHT INFRINGEMENT

DEMAND FOR JURY TRIAL

Plaintiff Oracle America, Inc., by and through its attorneys, alleges as follows:

PARTIES

1. Oracle America, Inc. ("Oracle America") is a corporation organized under the laws of the State of Delaware with its principal place of business at 500 Oracle Parkway, Redwood City, California 94065. Oracle America does business in the Northern District of California.

2. Upon information and belief, Defendant Google, Inc. ("Google") is a corporation organized under the laws of the State of Delaware with its principal place of business at 1600 Amphitheatre Parkway, Mountain View, California 94043. Google does business in the Northern District of California.

JURISDICTION AND VENUE

3. This is an action for patent and copyright infringement arising under the patent and copyright laws of the United States, Titles 35 and 17, United States Code. Jurisdiction as to these claims is conferred on this Court by 28 U.S.C. §§ 1331 and 1338(a).

4. Venue is proper in the Northern District of California under 28 U.S.C. §§ 1391 and 1400(b).

5. This Court has personal jurisdiction over Google. Google has conducted and does conduct business within the State of California and within this judicial district.

6. Google, directly or through intermediaries, makes, distributes, offers for sale or license, sells or licenses, and advertises its products and services in the United States, the State of California, and the Northern District of California.

INTRADISTRICT ASSIGNMENT

7. This is an Intellectual Property Action to be assigned on a district-wide basis pursuant to Civil Local Rule 3-2(c).

BACKGROUND

8. Oracle Corporation ("Oracle") is one of the world's leading technology companies, providing complete, open, and integrated business software and hardware systems. On January 27, 2010, Oracle acquired Sun Microsystems, Inc. ("Sun"). Sun is now Oracle America, a

subsidiary of Oracle. Oracle America continues to hold all of Sun's interest, rights, and title to the patents and copyrights at issue in this litigation.

9. One of the most important technologies Oracle acquired with Sun was the Java platform. The Java platform, which includes code and other documentation and materials, was developed by Sun and first released in 1995. The Java platform is a bundle of related programs, specifications, reference implementations, and developer tools and resources that allow a user to deploy applications written in the Java programming language on servers, desktops, mobile devices, and other devices. The Java platform is especially useful in that it insulates applications from dependencies on particular processors or operating systems. To date, the Java platform has attracted more than 6.5 million software developers. It is used in every major industry segment and has a ubiquitous presence in a wide range of computers, networks, and devices, including cellular telephones and other mobile devices. Sun's development of the Java platform resulted in many computing innovations and the issuance to Sun of a substantial number of important patents.

10. Oracle America is the owner by assignment of United States Patents Nos. 6,125,447; 6,192,476; 5,966,702; 7,426,720; RE38,104; 6,910,205; and 6,061,520, originally issued to Sun. True and correct copies of the patents at issue in this litigation are included as Exhibits A-G.

11. Oracle America owns copyrights in the code, documentation, specifications, libraries, and other materials that comprise the Java platform. Oracle America's Java-related copyrights are registered with the United States Copyright Office, including those attached as Exhibit H.

12. Google's Android competes with Oracle America's Java as an operating system software platform for cellular telephones and other mobile devices. The Android operating system software "stack" consists of Java applications running on a Java-based object-oriented application framework, and core libraries running on a "Dalvik" virtual machine (VM) that features just-in-time (JIT) compilation. Google actively distributes Android (including without

limitation the Dalvik VM and the Android software development kit) and promotes its use by manufacturers of products and applications.

13. Android (including without limitation the Dalvik VM and the Android software development kit) and devices that operate Android infringe one or more claims of each of United States Patents Nos. 6,125,447; 6,192,476; 5,966,702; 7,426,720; RE38,104; 6,910,205; and 6,061,520.

14. On information and belief, Google has been aware of Sun's patent portfolio, including the patents at issue, since the middle of this decade, when Google hired certain former Sun Java engineers.

15. On information and belief, Google has purposefully, actively, and voluntarily distributed Android and related applications, devices, platforms, and services with the expectation that they will be purchased, used, or licensed by consumers in the Northern District of California. Android has been and continues to be purchased, used, and licensed by consumers in the Northern District of California. Google has thus committed acts of patent infringement within the State of California and, particularly, within the Northern District of California. By purposefully and voluntarily distributing one or more of its infringing products and services, Google has injured Oracle America and is thus liable to Oracle America for infringement of the patents at issue in this litigation pursuant to 35 U.S.C. § 271.

COUNT I

(Infringement of the '447 Patent)

16. Oracle America hereby restates and realleges the allegations set forth in paragraphs 1 through 15 above and incorporates them by reference.

17. On September, 26, 2000, United States Patent No. 6,125,447, ("the '447 patent") entitled "Protection Domains To Provide Security In A Computer System" was duly and legally issued to Sun by the United States Patent and Trademark Office. Oracle America is the owner of the entire right, title, and interest in and to the '447 patent. A true and correct copy of the '447 patent is attached as Exhibit A to this Complaint.

18. Google actively and knowingly has infringed and is infringing the '447 patent with knowledge of Oracle America's patent rights and without reasonable basis for believing that Google's conduct is lawful. Google has also induced and contributed to the infringement of the '447 patent by purchasers, licensees, and users of Android, and is continuing to induce and contribute to the infringement of the '447 patent by purchasers, licensees, and users of Android. Google's acts of infringement have been and continue to be willful, deliberate, and in reckless disregard of Oracle America's patent rights. Google is thus liable to Oracle America for infringement of the '447 patent pursuant to 35 U.S.C. § 271.

COUNT II

(Infringement of the '476 Patent)

19. Oracle America hereby restates and realleges the allegations set forth in paragraphs 1 through 15 above and incorporates them by reference.

20. On February 20, 2000, United States Patent No. 6,192,476, ("the '476 patent") entitled "Controlling Access To A Resource" was duly and legally issued to Sun by the United States Patent and Trademark Office. Oracle America is the owner of the entire right, title, and interest in and to the '476 patent. A true and correct copy of the '476 patent is attached as Exhibit B to this Complaint.

21. Google actively and knowingly has infringed and is infringing the '476 patent with knowledge of Oracle America's patent rights and without reasonable basis for believing that Google's conduct is lawful. Google has also induced and contributed to the infringement of the '476 patent by purchasers, licensees, and users of Android, and is continuing to induce and contribute to the infringement of the '476 patent by purchasers, licensees, and users of Android. Google's acts of infringement have been and continue to be willful, deliberate, and in reckless disregard of Oracle America's patent rights. Google is thus liable to Oracle America for infringement of the '476 patent pursuant to 35 U.S.C. § 271.

COUNT III

(Infringement of the '702 Patent)

22. Oracle America hereby restates and realleges the allegations set forth in paragraphs 1 through 15 above and incorporates them by reference.

23. On October 12, 1999, United States Patent No. 5,966,702, ("the '702 patent") entitled "Method And Apparatus For Preprocessing And Packaging Class Files" was duly and legally issued to Sun by the United States Patent and Trademark Office. Oracle America is the owner of the entire right, title, and interest in and to the '702 patent. A true and correct copy of the '702 patent is attached as Exhibit C to this Complaint.

24. Google actively and knowingly has infringed and is infringing the '702 patent with knowledge of Oracle America's patent rights and without reasonable basis for believing that Google's conduct is lawful. Google has also induced and contributed to the infringement of the '702 patent by purchasers, licensees, and users of Android, and is continuing to induce and contribute to the infringement of the '702 patent by purchasers, licensees, and users of Android. Google's acts of infringement have been and continue to be willful, deliberate, and in reckless disregard of Oracle America's patent rights. Google is thus liable to Oracle America for infringement of the '702 patent pursuant to 35 U.S.C. § 271.

COUNT IV

(Infringement of the '720 Patent)

25. Oracle America hereby restates and realleges the allegations set forth in paragraphs 1 through 15 above and incorporates them by reference.

26. On September 16, 2008, United States Patent No. 7,426,720, ("the '720 patent") entitled "System And Method For Dynamic Preloading Of Classes Through Memory Space Cloning Of A Master Runtime System Process" was duly and legally issued to Sun by the United States Patent and Trademark Office. Oracle America is the owner of the entire right, title, and interest in and to the '720 patent. A true and correct copy of the '720 patent is attached as Exhibit D to this Complaint.

27. Google actively and knowingly has infringed and is infringing the '720 patent with knowledge of Oracle America's patent rights and without reasonable basis for believing that Google's conduct is lawful. Google has also induced and contributed to the infringement of the '720 patent by purchasers, licensees, and users of Android, and is continuing to induce and contribute to the infringement of the '720 patent by purchasers, licensees, and users of Android. Google's acts of infringement have been and continue to be willful, deliberate, and in reckless disregard of Oracle America's patent rights. Google is thus liable to Oracle America for infringement of the '720 patent pursuant to 35 U.S.C. § 271.

COUNT V

(Infringement of the '104 Patent)

28. Oracle America hereby restates and realleges the allegations set forth in paragraphs 1 through 15 above and incorporates them by reference.

29. On April 29, 2003, United States Patent No. RE38,104, ("the '104 patent") entitled "Method And Apparatus For Resolving Data References In Generate Code" was duly and legally issued to Sun by the United States Patent and Trademark Office. Oracle America is the owner of the entire right, title, and interest in and to the '104 patent. A true and correct copy of the '104 patent is attached as Exhibit E to this Complaint.

30. Google actively and knowingly has infringed and is infringing the '104 patent with knowledge of Oracle America's patent rights and without reasonable basis for believing that Google's conduct is lawful. Google has also induced and contributed to the infringement of the '104 patent by purchasers, licensees, and users of Android, and is continuing to induce and contribute to the infringement of the '104 patent by purchasers, licensees, and users of Android. Google's acts of infringement have been and continue to be willful, deliberate, and in reckless disregard of Oracle America's patent rights. Google is thus liable to Oracle America for infringement of the '104 patent pursuant to 35 U.S.C. § 271.

COUNT VI

(Infringement of the '205 Patent)

31. Oracle America hereby restates and realleges the allegations set forth in paragraphs 1 through 15 above and incorporates them by reference.

32. On June 21, 2005, United States Patent No. 6,910,205, ("the '205 patent") entitled "Interpreting Functions Utilizing A Hybrid Of Virtual And Native Machine Instructions" was duly and legally issued to Sun by the United States Patent and Trademark Office. Oracle America is the owner of the entire right, title, and interest in and to the '205 patent. A true and correct copy of the '205 patent is attached as Exhibit F to this Complaint.

33. Google actively and knowingly has infringed and is infringing the '205 patent with knowledge of Oracle America's patent rights and without reasonable basis for believing that Google's conduct is lawful. Google has also induced and contributed to the infringement of the '205 patent by purchasers, licensees, and users of Android, and is continuing to induce and contribute to the infringement of the '205 patent by purchasers, licensees, and users of Android. Google's acts of infringement have been and continue to be willful, deliberate, and in reckless disregard of Oracle America's patent rights. Google is thus liable to Oracle America for infringement of the '205 patent pursuant to 35 U.S.C. § 271.

COUNT VII

(Infringement of the '520 Patent)

34. Oracle America hereby restates and realleges the allegations set forth in paragraphs 1 through 15 above and incorporates them by reference.

35. On May 9, 2000, United States Patent No. 6,061,520, ("the '520 patent") entitled "Method And System for Performing Static Initialization" was duly and legally issued to Sun by the United States Patent and Trademark Office. Oracle America is the owner of the entire right, title, and interest in and to the '520 patent. A true and correct copy of the '520 patent is attached as Exhibit G to this Complaint.

36. Google actively and knowingly has infringed and is infringing the '520 patent with knowledge of Oracle America's patent rights and without reasonable basis for believing that

Google's conduct is lawful. Google has also induced and contributed to the infringement of the '520 patent by purchasers, licensees, and users of Android, and is continuing to induce and contribute to the infringement of the '520 patent by purchasers, licensees, and users of Android. Google's acts of infringement have been and continue to be willful, deliberate, and in reckless disregard of Oracle America's patent rights. Google is thus liable to Oracle America for infringement of the '520 patent pursuant to 35 U.S.C. § 271.

COUNT VIII

(Copyright Infringement)

37. Oracle America hereby restates and realleges the allegations set forth in paragraphs 1 through 15 above and incorporates them by reference.

38. The Java platform contains a substantial amount of original material (including without limitation code, specifications, documentation and other materials) that is copyrightable subject matter under the Copyright Act, 17 U.S.C. § 101 *et seq*.

39. Without consent, authorization, approval, or license, Google knowingly, willingly, and unlawfully copied, prepared, published, and distributed Oracle America's copyrighted work, portions thereof, or derivative works and continues to do so. Google's Android infringes Oracle America's copyrights in Java and Google is not licensed to do so.

40. On information and belief, users of Android, including device manufacturers, must obtain and use copyrightable portions of the Java platform or works derived therefrom to manufacture and use functioning Android devices. Such use is not licensed. Google has thus induced, caused, and materially contributed to the infringing acts of others by encouraging, inducing, allowing and assisting others to use, copy, and distribute Oracle America's copyrightable works, and works derived therefrom.

41. On information and belief, Google's direct and induced infringements are and have been knowing and willful.

42. By this unlawful copying, use, and distribution, Google has violated Oracle America's exclusive rights under 17 U.S.C. § 106.

43. Google has realized unjust profits, gains and advantages as a proximate result of its infringement.

44. Google will continue to realize unjust profits, gains and advantages as a proximate result of its infringement as long as such infringement is permitted to continue.

45. Oracle America is entitled to an injunction restraining Google from engaging in any further such acts in violation of the United States copyright laws. Unless Google is enjoined and prohibited from infringing Oracle America's copyrights, inducing others to infringe Oracle America's copyrights, and unless all infringing products and advertising materials are seized, Google will continue to intentionally infringe and induce infringement of Oracle America's registered copyrights.

46. As a direct and proximate result of Google's direct and indirect willful copyright infringement, Oracle America has suffered, and will continue to suffer, monetary loss to its business, reputation, and goodwill. Oracle America is entitled to recover from Google, in amounts to be determined at trial, the damages sustained and will sustain, and any gains, profits, and advantages obtained by Google as a result of Google's acts of infringement and Google's use and publication of the copied materials.

PRAYER FOR RELIEF

WHEREFORE, Oracle America prays for judgment as follows:

A. Entry of judgment holding Google liable for infringement of the patents and copyrights at issue in this litigation;

B. An order permanently enjoining Google, its officers, agents, servants, employees, attorneys and affiliated companies, its assigns and successors in interest, and those persons in active concert or participation with it, from continued acts of infringement of the patents and copyrights at issue in this litigation;

C. An order that all copies made or used in violation of Oracle America's copyrights, and all means by which such copies may be reproduced, be impounded and destroyed or otherwise reasonably disposed of;

D. An order awarding Oracle America statutory damages and damages according to

proof resulting from Google's infringement of the patents and copyrights at issue in this litigation, together with prejudgment and post-judgment interest;

E. Trebling of damages under 35 U.S.C. § 284 in view of the willful and deliberate nature of Google's infringement of the patents at issue in this litigation;

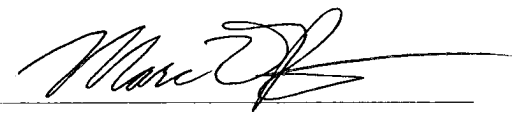
F. An order awarding Oracle America its costs and attorney's fees under 35 U.S.C. § 285 and 17 U.S.C. § 505; and

G. Any and all other legal and equitable relief as may be available under law and which the court may deem proper.

DEMAND FOR A JURY TRIAL

Oracle America demands a jury trial for all issues so triable.

Dated: August 12, 2010

By: 

MORRISON & FOERSTER LLP
MICHAEL A. JACOBS (Bar No. 111664)
mjacobs@mofo.com
MARC DAVID PETERS (Bar No. 211725)
mdpeters@mofo.com
755 Page Mill Road
Palo Alto, CA 94304-1018
Telephone: (650) 813-5600
Facsimile (650) 494-0792

BOIES, SCHILLER & FLEXNER LLP
DAVID BOIES (*Pro Hac Vice* Pending)
dboies@bsfllp.com
333 Main Street
Armonk, NY 10504
Telephone: (914) 749-8200
Facsimile: (914) 749-8300
STEVEN C. HOLTZMAN (Bar No. 144177)
sholtzman@bsfllp.com
1999 Harrison St., Suite 900
Oakland, CA 94612
Telephone (510) 874-1000
Facsimile: (510) 874-1460

ORACLE CORPORATION
MATTHEW M. SARBORARIA
(Bar No. 211600)
matthew.sarboraria@oracle.com
500 Oracle Parkway
Redwood City, CA 94065
Telephone: (650) 506-5200
Facsimile: (650) 506-7114

Attorneys for Plaintiff
ORACLE AMERICA, INC.

EXHIBIT A

US006125447A

United States Patent [19]

Gong

[11] Patent Number: 6,125,447

[45] Date of Patent: Sep. 26, 2000

[54] **PROTECTION DOMAINS TO PROVIDE SECURITY IN A COMPUTER SYSTEM**

[75] Inventor: **Li Gong**, Menlo Park, Calif.

[73] Assignee: **Sun Microsystems, Inc.**, Mountain View, Calif.

[21] Appl. No.: **08/988,439**

[22] Filed: **Dec. 11, 1997**

[51] **Int. Cl.**[7] **H04L 9/00**

[52] **U.S. Cl.** **713/201**; 713/154

[58] **Field of Search** 713/200, 201–202, 713/151, 152, 153, 154–168, 169; 709/229, 303; 395/704; 714/38, 48; 707/103, 9, 10; 380/4

[56] **References Cited**

U.S. PATENT DOCUMENTS

5,311,591	5/1994	Fischer	380/4
5,720,033	2/1998	Deo	713/200
5,758,153	5/1998	Atsatt et al.	395/614
5,841,870	11/1998	Fieres et al.	380/25
5,845,129	12/1998	Wendorf et al.	395/726
5,892,904	4/1999	Atkinson et al.	713/201

FOREIGN PATENT DOCUMENTS

2259590A	3/1993	WIPO	G06F 9/44
2308688A	7/1997	WIPO	G06F 12/14

OTHER PUBLICATIONS

Gong Li, et al.: "Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java™ Development Kit 1.2", Proceedings of the Usenix Symposium on Internet Technologies and Systems, Monterey, CA, USA, 8–11 Dec. 1997, ISBN 1–880446–91–S, 1997, Berkeley, CA, USA, Usenix Assoc., USA, pp. 103–112, XP002100907.

Wallach, D. S., et al.: "Extensible Security Architectures for Java", 16th ACM Symposium on Operating Systems Principles, Sain Malo, France, 5–8 Oct. 1997, ISSN 0163–5980, Operating Systems Review, Dec. 1997, ACM, USA, pp. 116–128, XP–002101681.

Dean, D., et al., "Java Security: From HotJava to Netscape and Beyond," Proceedings of the 1996 IEEE Symposium on Security and Privacy, Oakland, CA, May 6–8, 1996.

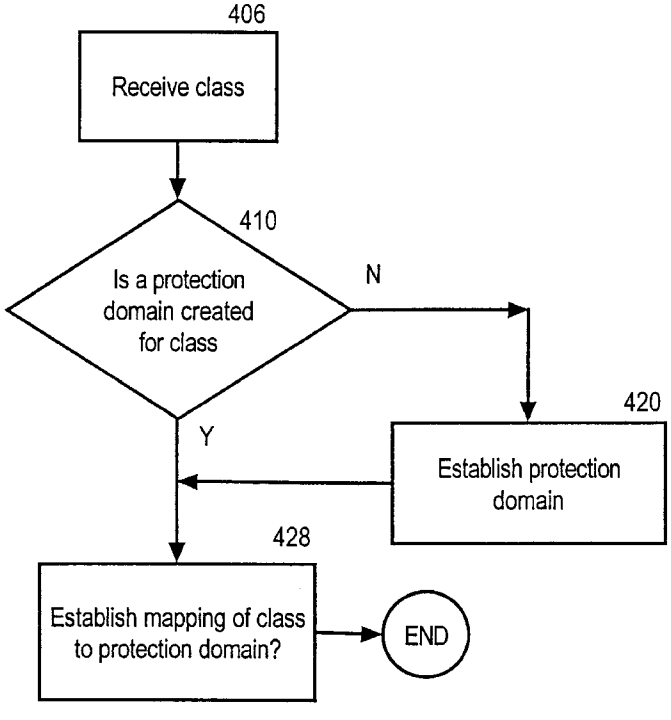
Hamilton, M.A., "Java and the Shift to Net–Centric Computing," Computer, vol. 29, No. 8, Aug., 1996.

Primary Examiner—Robert W. Beausoliel, Jr.
Assistant Examiner—Scott T. Baderman
Attorney, Agent, or Firm—McDermott, Will & Emery

[57] **ABSTRACT**

A method and apparatus are provided for maintaining and enforcing security rules using protection domains. As new code arrives at a computer, a determination is assigned to a protection domain based on the source from which the code is received. The protection domain establishes the permissions that apply to the code. In embodiments where the code to be executed by the computer belongs to object classes, an association is established between the protection domains and the classes of objects. When an object requests an action, a determination is made as to whether the action is permitted based on the class to which the object belongs and the association between classes and protection domains.

24 Claims, 6 Drawing Sheets



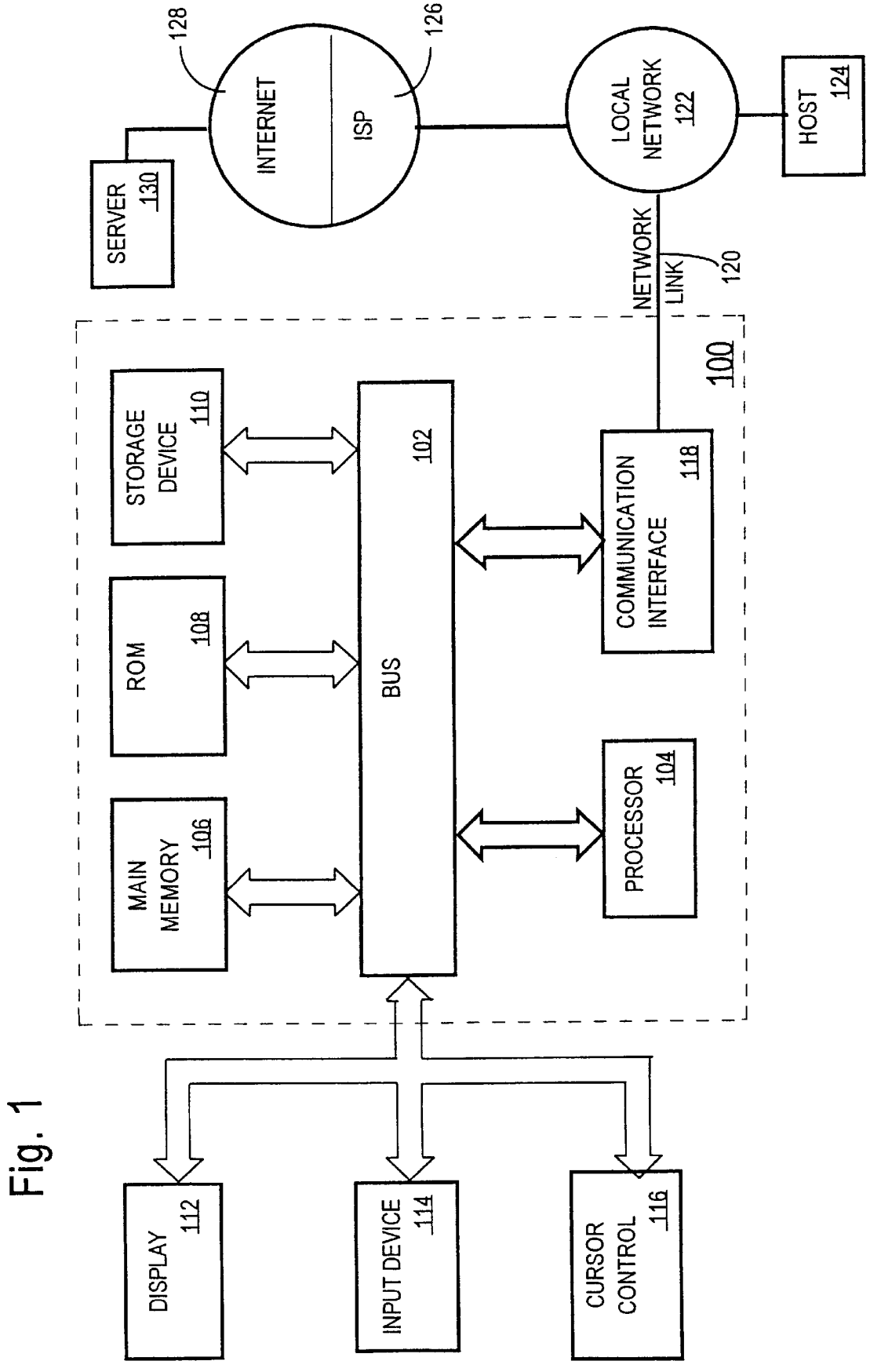

128
INTERNET
ISP
126
LOCAL NETWORK 122
HOST 124
SERVER 130
NETWORK LINK 120
100
STORAGE DEVICE 110
ROM 108
MAIN MEMORY 106
BUS 102
COMMUNICATION INTERFACE 118
PROCESSOR 104
DISPLAY 112
INPUT DEVICE 114
CURSOR CONTROL 116

Fig. 1

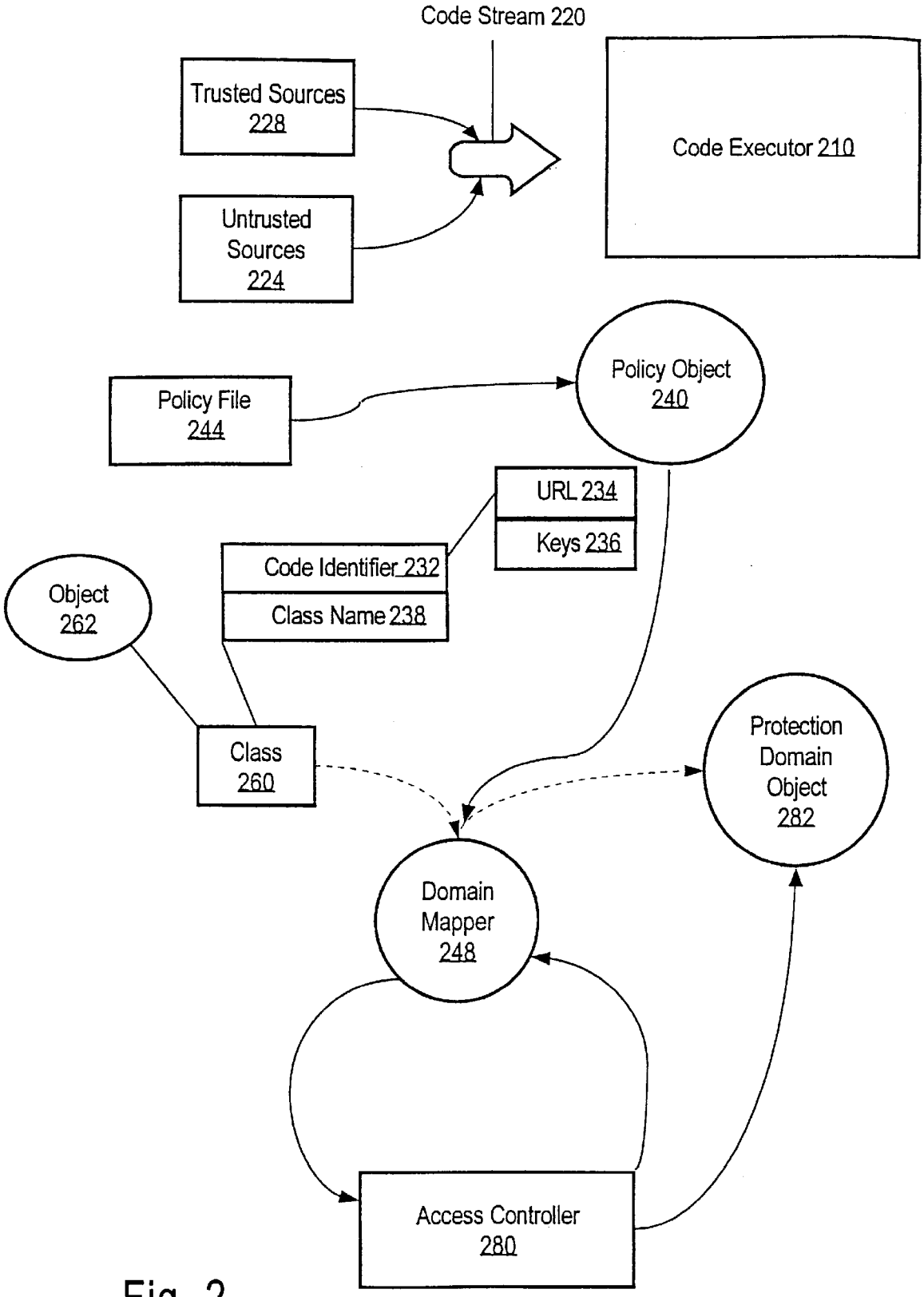

Code Stream 220
Trusted Sources 228
Untrusted Sources 224
Code Executor 210
Policy Object 240
Policy File 244
URL 234
Keys 236
Code Identifier 232
Class Name 238
Object 262
Class 260
Protection Domain Object 282
Domain Mapper 248
Access Controller 280

Fig. 2

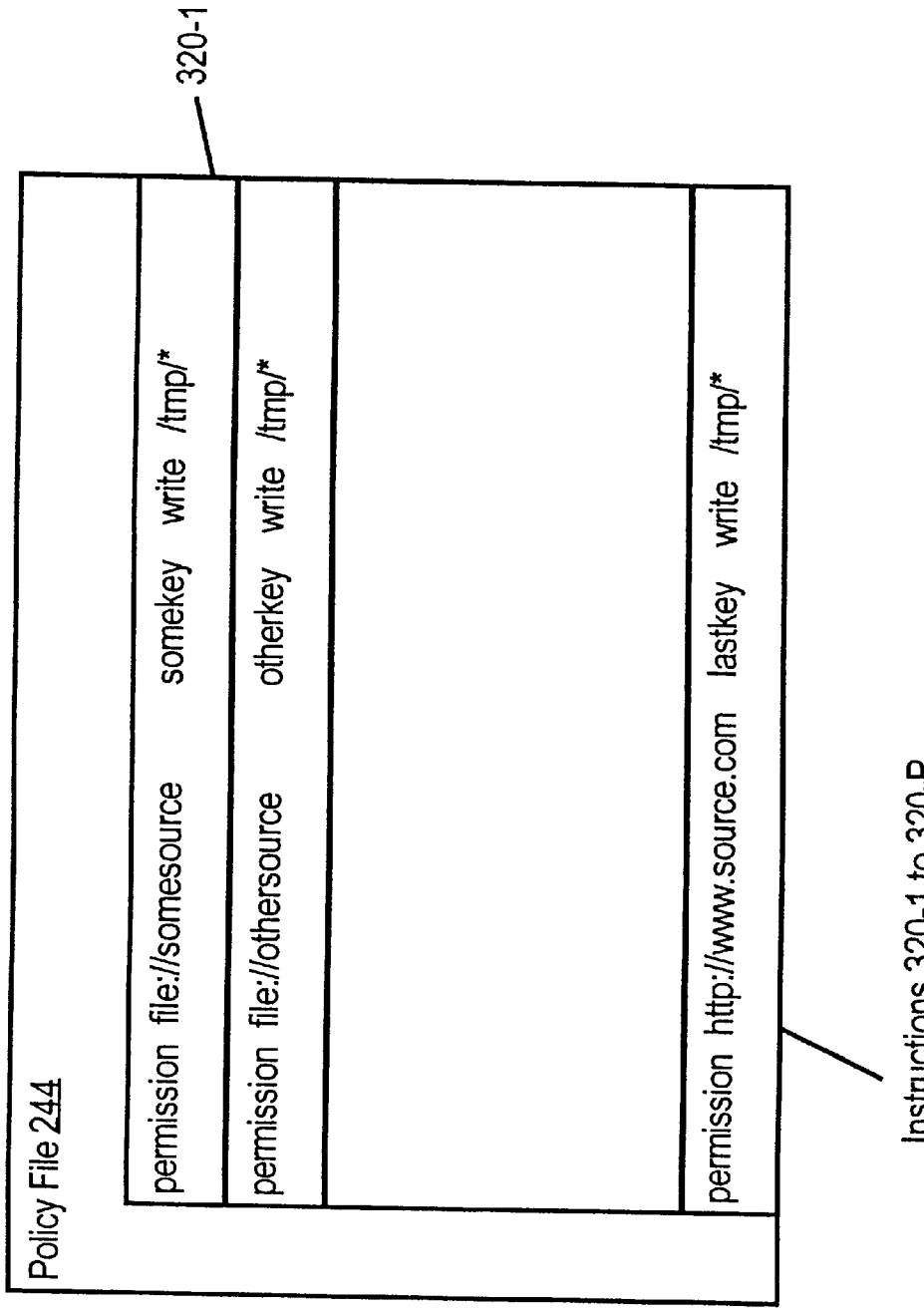

320-1
Policy File 244
permission file://somesource somekey write /tmp/*
permission file://othersource otherkey write /tmp/*
permission http://www.source.com lastkey write /tmp/*
Instructions 320-1 to 320-P

Fig. 3

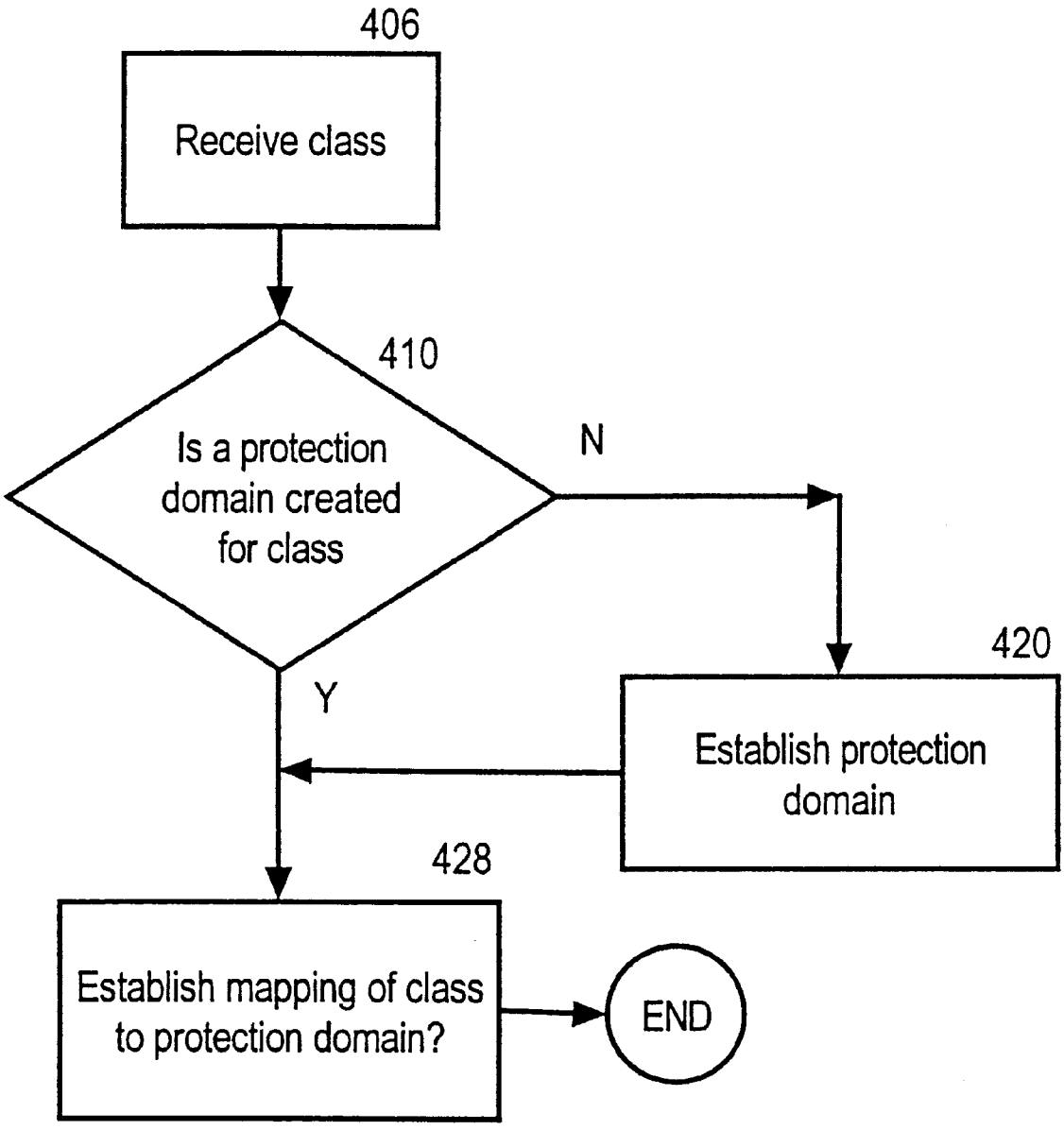

406
Receive class
410
Is a protection domain created for class
N
Y
420
Establish protection domain
428
Establish mapping of class to protection domain?
END

Fig. 4

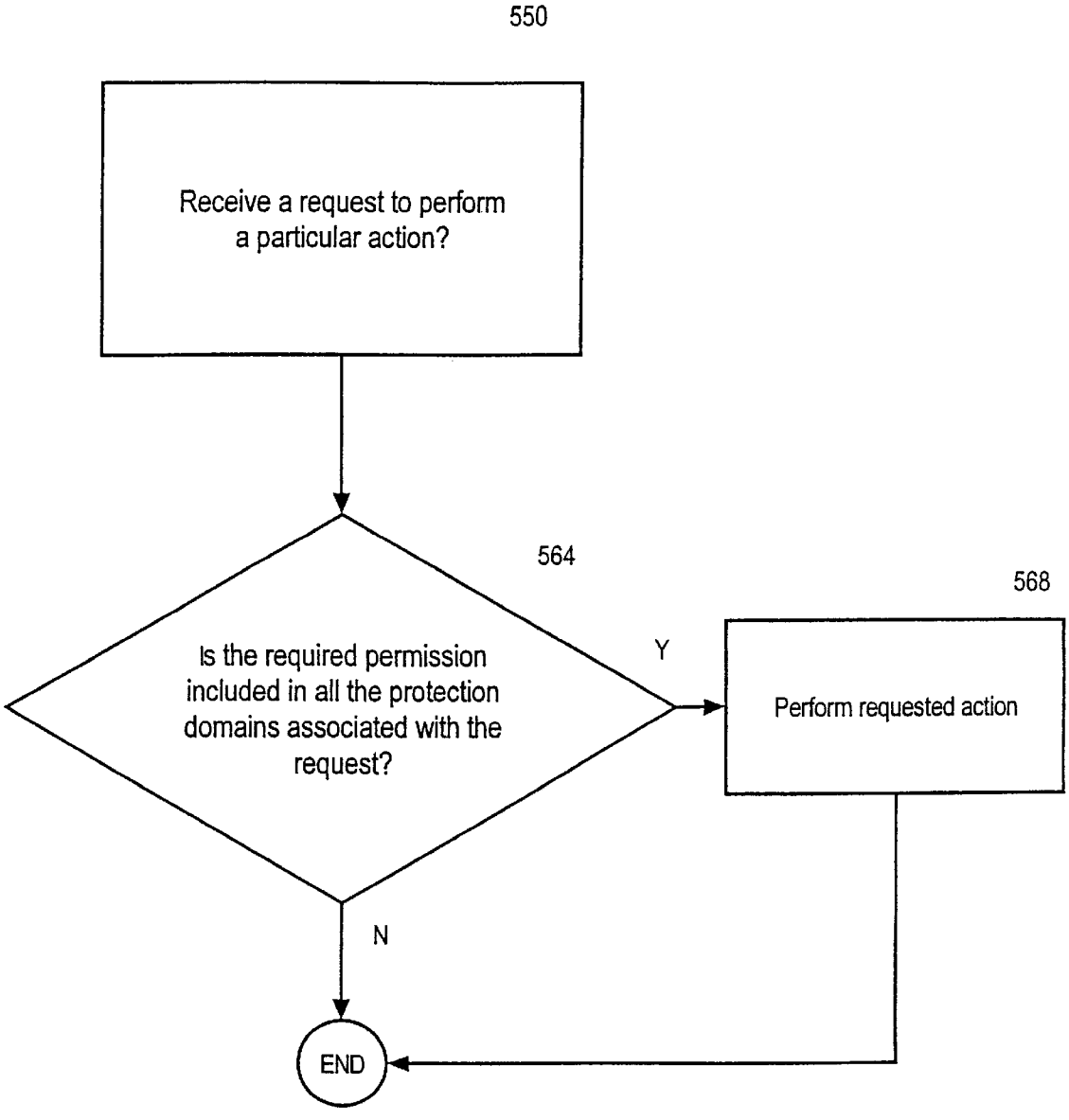

550
Receive a request to perform a particular action?
564
Is the required permission included in all the protection domains associated with the request?
Y
568
Perform requested action
N
END

Fig. 5

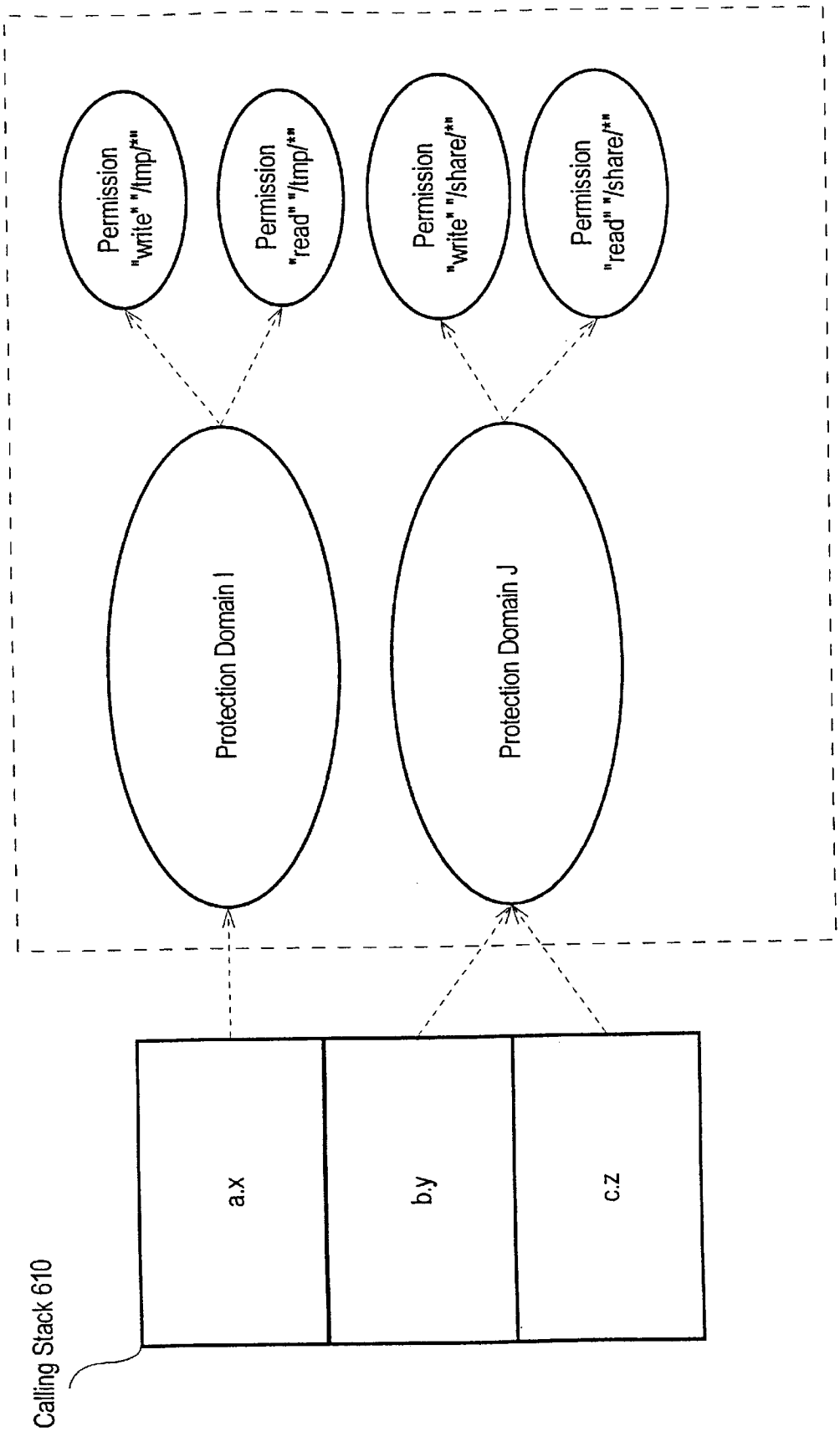

Permission "write" "/tmp/*"
Permission "read" "/tmp/*"
Permission "write" "/share/*"
Permission "read" "/share/*"
Protection Domain I
Protection Domain J
Security Policy
a.x
b.y
c.z
Calling Stack 610

Fig. 6

PROTECTION DOMAINS TO PROVIDE SECURITY IN A COMPUTER SYSTEM

RELATED APPLICATIONS

The present application is related to U.S. patent application Ser. No. 08/988,857, entitled "TYPED, PARAMETERIZED, AND EXTENSIBLE ACCESS CONTROL PERMISSIONS", filed by Li Gong, on the equal day herewith, now U.S. Pat. No. 6,047,377 the contents of which are incorporated herein by reference.

The present application is related to U.S. patent application Ser. No. 08/988,431, entitled "CONTROLLING ACCESS TO A RESOURCE", filed by Li Gong, on the equal day herewith, the contents of which are incorporated herein by reference.

The present application is related to U.S. patent application Ser. No. 08/988,660, entitled "SECURE CLASS RESOLUTION, LOADING, AND DEFINITION", filed by Li Gong, on the equal day herewith, now U.S. Pat. No. 6,044,467, the contents of which are incorporated herein by reference.

FIELD OF THE INVENTION

The present invention relates to security mechanisms in a computer system.

BACKGROUND OF THE INVENTION

As the use of computer systems grows, organizations are becoming increasingly reliant upon them. A malfunction in the computer system can severely hamper the operation of such organizations. Thus organizations that use computer systems are vulnerable to users who may intentionally or unintentionally cause the computer system to malfunction.

One way to compromise the security of a computer system is to cause the computer system to execute software that performs harmful actions on the computer system. There are various types of security measures that may be used to prevent a computer system from executing harmful software. One example is to check all software executed by the computer system with a "virus" checker. However, virus checkers only search for very specific software instructions. Many methods of using software to tamper with a computer's resources would not be detected by a virus checker.

Another very common measure used to prevent the execution of software that tampers with a computer's resources is the "trusted developers approach". According to the trusted developers approach, system administrators limit the software that a computer system can access to only software developed by trusted software developers. Such trusted developers may include, for example, well know vendors or in-house developers.

Fundamental to the trusted developers approach is the idea that computer programs are created by developers, and that some developers can be trusted to not have produced software that compromises security. Also fundamental to the trusted developers approach is the notion that a computer system will only execute programs that are stored at locations that are under control of the system administrators.

Recently developed methods of running applications involve the automatic and immediate execution of software code loaded from remote sources over a network. When the network includes remote sources that are outside the control of system administrators, the trusted developers approach does not work.

One attempt to adapt the trusted developers approach to systems that can execute code from remote sources is

referred to as the sand box method. The sand box method allows all code to be executed, but places restrictions on remote code. Specifically, the sand box method permits all trusted code full access to computer system's resources and all remote code limited access to a computer system's resources. Trusted code is usually stored locally on the computer system under the direct control of the owners or administrators of the computer system, who are accountable for the security of the trusted code.

One drawback to the sandbox approach is that the approach is not very granular. The sandbox approach is not very granular because all remote code is restricted to the same limited set of resources. Very often, there is a need to permit remote code from one source access to one set of computer resources while permitting remote code from another source access to another set of computer resources. For example, there may be a need to limit access to a one set of files associated with one bank to remote code loaded over a network from a source associated with that one bank, and limit access to another set of files associated with another bank to remote code loaded over a network from a source associated with the other bank.

Providing security measures that allow more granularity than the sand box method involves establishing a complex set of relationships between principals and permissions. A "principal" is an entity in the computer system to which permissions are granted. Examples of principals include processes, objects and threads. A "permission" is an authorization by the computer system that allows a principal to perform a particular action or function.

Establishing sets of permissions for principals that may be received from multiple sources on a vast network, such as the Internet, typically requires developing complex security software. After such security software is developed, it must often be changed in order to meet changing security requirements. Often, changing security requirements entail modifying permissions or creating new kinds of permissions. Typically, the security software of a computer system must be reprogrammed to incorporate these new kinds of permissions. Programming security software requires substantial effort and in-depth knowledge about a computer's security mechanisms and a computer's architecture.

Based on the foregoing, it is clearly desirable to develop a method which reduces the effort and in-depth knowledge required to modify permissions established for the sources of code being executed by a computer system. It is further desirable to develop a method which reduces the effort and in-depth knowledge required to create new permissions.

SUMMARY OF THE INVENTION

A method and system are provided for implementing security policies within a computer system. The security mechanism makes use of structures referred to herein as "protection domains" to organize, represent and maintain the security policies that apply to the computer system.

According to one aspect of the invention, protection domains are established based on policy data, where each protection domain is associated with zero or more permissions. An association is established between the protection domains and classes of objects (i.e. instantiations of the classes) that may be invoked by the computer system. When an object requests an action, a determination is made as to whether the action is permitted for that object. The determination is based on the association between the protection domains and the classes. For example, based on policy data, an association between Class CA and protection domain PA,

and class CB and protection domain PB is established. Protection domain PA and protection domain PB are each associated with a set of permissions. Object OA belongs (i.e. is an instantiation of) to class CA. When object OA requests an action, a determination of whether the action is permitted is based on the permissions associated with protection domain PA, because protection domain PA is associated class CA.

According to another aspect of the invention, each protection domain and class is associated with a code identifier. An association between the protection domains and the classes is based on the code identifier. The code identifier may contain data describing the source of code that defines a class, a set of public cryptographic keys associated with the source of code, or other information which describes the source of code, or any combination thereof. A "source of code" is an entity from which computer instructions are received. Examples of sources of code include a file or persistent object stored on a data server connected over a network, a FLASH_EPROM reader that reads instructions stored on a FLASH_EPROM, or a set of system libraries.

The association between protection domains and code identifiers is typically recorded in data persistently stored in the computer system. The data associates code identifiers with one or more permissions. For example, the code identifier associated with class CA has a value "A.host/fileA", indicating that the source of class CA is A.hostA/fileA, a file on host A connected over the Internet. A security policy file associates permission **1** and permission **2** with A.host/A.fileA. Protection domain A, which is associated with a code identifier having a value of "A.hostA/fileA", contains permission **1** and permission **2**. Protection domain A is associated class CA, because protection domain A and protection domain are associated with the same code identifier value, "A.host/fileA".

According to another aspect of the invention, when executing code causes a request for an action, a determination is made as to whether the action is permitted. The determination is based on the source of code of the code causing the request and the association between protection domains and sources of code executed by the computer system. According to another aspect of the invention, the association between protection domains and the sources of code is also based on public cryptographic keys associated with the sources of code. For example, code causing a request is from source A.host/fileA, and protection domain A is associated is A.host/fileA. A determination of whether the request may be honored is based on the protection domain A, and in particular, to the permissions associated with protection domain A.

BRIEF DESCRIPTION OF THE DRAWINGS

The present invention is illustrated by way of example, and not by way of limitation, in the figures of the accompanying drawings and in which like reference numerals refer to similar elements and in which:

FIG. **1** is a block diagram of a computer system on which the present invention may be implemented in accordance with an embodiment of the present invention;

FIG. **2** is a block diagram showing the objects used by a code executor implementing protection domains in accordance with an embodiment of the present invention;

FIG. **3** is block diagram showing an exemplary policy file in accordance with an embodiment of the present invention;

FIG. **4** is a flow chart showing the steps involved in implementing protection domains in accordance with an embodiment of the present invention;

FIG. **5** is a flow chart showing the steps followed by an access controller using an implementation of protection domains in accordance with an embodiment of the present invention; and

FIG. **6** is a block diagram showing a call stack representing objects associated with protection domains and permissions in accordance with an embodiment of the present invention.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

A method and apparatus for providing security through the use of protection domains is described. In the following description, for the purposes of explanation, numerous specific details are set forth in order to provide a thorough understanding of the present invention. It will be apparent, however, to one skilled in the art that the present invention may be practiced without these specific details. In other instances, well-known structures and devices are shown in block diagram form in order to avoid unnecessarily obscuring the present invention.

Hardware Overview

FIG. **1** is a block diagram which illustrates a computer system **100** upon which an embodiment of the invention may be implemented. Computer system **100** includes a bus **102** or other communication mechanism for communicating information, and a processor **104** coupled with bus **102** for processing information. Processor **104** generally represents one or more processors capable of executing instructions that conform to a particular instruction set. Computer system **100** also includes a main memory **106**, such as a random access memory (RAM) or other dynamic storage device, coupled to bus **102** for storing information and instructions to be executed by processor **104**. Main memory **106** also may be used for storing temporary variables or other intermediate information during execution of instructions by processor **104**. Computer system **100** also includes a read only memory (ROM) **108** or other static storage device coupled to bus **102** for storing static information and instructions for processor **104**. A storage device **110**, such as a magnetic disk or optical disk, is also provided and coupled to bus **102** for storing information and instructions.

Computer system **100** may also be coupled via bus **102** to a display **112**, such as a cathode ray tube (CRT), for displaying information to a computer user. An input device **114**, including alphanumeric and other keys, is also provided and coupled to bus **102** for communicating information and command selections to processor **104**. Another type of user input device is cursor control **116**, such as a mouse, a trackball, or cursor direction keys for communicating direction information and command selections to processor **104** and for controlling cursor movement on display **112**. This input device typically has two degrees of freedom in two axes, a first axis (e.g., x) and a second axis (e.g., y), which allows the device to specify positions in a plane.

The invention is related to the use of computer system **100** for the implementation of protection domains. According to one embodiment of the invention, the implementation of protection domains is provided by computer system **100** in response to processor **104** executing sequences of instructions contained in main memory **106**. Such instructions may be read into main memory **106** from another computer-readable medium, such as storage device **110**. In alternative embodiments, hard-wired circuitry may be used in place of or in combination with software instructions to implement

the invention. Thus, embodiments of the invention are not limited to any specific combination of hardware circuitry and software.

The term "computer-readable medium" as used herein refers to any medium that participates in providing instructions to processor **104** for execution. Such a medium may take many forms, including but not limited to, non-volatile media, volatile media, and transmission media. Non-volatile media includes, for example, optical or magnetic disks, such as storage device **110**. Volatile media includes dynamic memory, such as main memory **106**. Transmission media includes coaxial cables, copper wire and fiber optics, including the wires that comprise bus **102**. Transmission media can also take the form of acoustic or light waves, such as those generated during radio-wave and infra-red data communications.

Common forms of computer-readable media include, for example, a floppy disk, a flexible disk, hard disk, magnetic tape, or any other magnetic medium, a CD-ROM, any other optical medium, punchcards, papertape, any other physical medium with patterns of holes, a RAM, a PROM, and EPROM, a FLASH-EPROM, any other memory chip or cartridge, a carrier wave as described hereinafter, or any other medium from which a computer can read.

Various forms of computer readable media may be involved in carrying one or more sequences of one or more instructions to processor **104** for execution. For example, the instructions may initially be carried on a magnetic disk of a remote computer. The remote computer can load the instructions into its dynamic memory and send the instructions over a telephone line using a modem. A modem local to computer system **100** can receive the data on the telephone line and use an infra-red transmitter to convert the data to an infra-red signal. An infra-red detector coupled to bus **102** can receive the data carried in the infra-red signal and place the data on bus **102**. Bus **102** carries the data to main memory **106**, from which processor **104** retrieves and executes the instructions. The instructions received by main memory **106** may optionally be stored on storage device **110** either before or after execution by processor **104**.

Computer **100** also includes a communication interface **118** coupled to bus **102**. Communication interface **118** provides a two-way data communication coupling to a network link **120** to a local network **122**. For example, if communication interface **118** is an integrated services digital network (ISDN) card or a modem, communication interface **118** provides a data communication connection to the corresponding type of telephone line. If communication interface **118** is a local area network (LAN) card, communication interface **118** provides a data communication connection to a compatible LAN. Wireless links are also possible. In any such implementation, communication interface **118** sends and receives electrical, electromagnetic or optical signals which carry digital data streams representing various types of information.

Network link **120** typically provides data communication through one or more networks to other data devices. For example, network link **120** may provide a connection through local network **122** to a host computer **124** or to data equipment operated by an Internet Service Provider (ISP) **126**. ISP **126** in turn provides data communication services through the world wide packet data communication network now commonly referred to as the "Internet" **128**. Local network **122** and Internet **128** both use electrical, electromagnetic or optical signals which carry digital data streams. The signals through the various networks and the signals on network link **120** and through communication interface **118**, which carry the digital data to and from computer **100** are exemplary forms of carrier waves transporting the information.

Computer **100** can send messages and receive data, including program code, through the network(s), network link **120** and communication interface **118**. In the Internet example, a server **130** might transmit a requested code for an application program through Internet **128**, ISP **126**, local network **122** and communication interface **118**.

The received code may be executed by processor **104** as it is received, and/or stored in storage device **110**, or other non-volatile storage for later execution. In this manner, computer **100** may obtain application code in the form of a carrier wave.

In alternative embodiments, hard-wired circuitry may be used in place of or in combination with software instructions to implement the present invention. Thus, the present invention is not limited to any specific combination of hardware circuitry and software.

Functional Overview

As mentioned above, systems that allow execution of software from remote sources present difficult security problems. The systems that have been developed to address those problems are complex, often requiring the use of elaborate permission rules to deal with principals received from numerous sources. As the security needs of the systems change, the permission rules must be updated by someone who understands the complexities of the system.

According to one aspect of the invention, the complexities associated with elaborate permission rules and systems are reduced by establishing dynamically constructing and establishing protection domains for code that arrives for execution on a computer system. The protection domains embody sets of permissions and are constructed based on policy information.

The use of the protection domains provides a relatively simple mechanism to implement otherwise complex security policies. As the security needs of a system changes, the security mechanism described herein allows easy modification to adapt to the changes, without requiring specialized knowledge of complex security-management techniques.

Exemplary Security Mechanism

An exemplary security mechanism illustrating the use of protection domains is shown in FIG. **2**. Referring to FIG. **2**, the exemplary security mechanism includes a policy file **244**, a policy object **240**, a domain mapper object **248**, an access controller **280**, and one or more protection domains **282**. The security mechanism is implemented using a code executor **210**.

Code executor **210** executes code which code executor **210** receives from code stream **220**. One example of a code executor is a Java virtual machine. A Java virtual machine interprets code called byte code. Byte code is code generated by a Java compiler from source files containing text. The Java virtual machine is described in detail in Tim Lindholm & Frank Yellin, *The Java Virtual Machine Specification* (1996).

For the purposes of explanation, it shall be assumed that code from code stream **220** is object oriented software. Consequently, the code is in the form of methods associated with objects that belong to classes. In response to instructions embodied by code executed by code executor **210**,

code executor **210** creates one or more objects **240**. An object is a record of data combined with the procedures and functions that manipulate the record. All objects belong to a class. Each object belonging to a class has the same fields ("attributes") and the same methods. The methods are the procedures, functions, or routines used to manipulate the object. An object is said to be an "instance" of the class to which the object belongs.

One or more class definitions are contained in code from code stream **220**. The fields and methods of the objects belonging to a class are defined by a class definition. These class definitions are used by code executor **210** to create objects which are instances of the classes defined by the class definitions.

These class definitions are generated from source code written by a programmer. For example, a programmer using a Java Development Kit enters source code that conforms to the Java programming language into a source file. The source code embodies class definitions and other instructions which are used to generate byte code which controls the execution of a code executor (i.e. a Java virtual machine). Techniques for defining classes and generating code executed by a code executor, such as a Java virtual machine, are well known to those skilled in the art.

Each class defined by a class definition from code stream **220** is associated with a class name **238** and a code source **236**. Code executor **210** maintains an association between a class and its class name and code source.

The code source may be a composite record containing a uniform resource locator ("URL") **234** and set of public cryptographic keys **236**. A URL identifies a particular source. The URL is a string used to uniquely identify any server connected to the world wide web. A URL may also be used to designate sources local to computer system **100**. Typically, the URL includes the designation of the file and directory of the file that is the source of the code stream that a server is providing.

A public cryptographic key, herein referred to as a key, is used to validate the digital signature which may be included in a file used to transport related code and data. Public cryptographic keys and digital signatures are described in Schneier, *Applied Crytography*, (1996). The keys may be contained in the file, may be contained in a database associating keys with sources (e.g. URLs), or be accessible using other possible alternative techniques.

A class may be associated with the digital signature associated with the file used to transport code defining the class, or the class definition of the class may be specifically associated with a digital signature. A class that is associated with a valid digital signature is referred to as being signed. Valid digital signatures are digital signatures that can be verified by known keys stored in a database. If a class is associated with a digital signature which can not be verified, or the class is not associated with any digital signature, the class is referred to as being unsigned. Unsigned classes may be associated with a default key. A key may be associated with a name, which may be used to look up the key in the database.

Trusted and Untrusted Sources

The source of code stream **220** may be from zero or more untrusted sources **224** or zero or more trusted sources **228**. Untrusted sources **224** and trusted sources **228** may be file servers, including file servers that are part of the World Wide Web network of servers connected to the Internet. An untrusted source is typically not under the direct control of the operators of computer system **100**. Code from untrusted sources is herein referred to as untrusted code.

Because untrusted code is considered to pose a high security risk, the set of computer resources that untrusted code may access is usually restricted to those which do not pose security threats. Code from a trusted source is code usually developed by trusted developers. Trusted code is considered to be reliable and pose much less security risk than remote code.

Software code which is loaded over the network from a remote source and immediately executed is herein referred to as remote code. Typically, a remote source is a computer system of another separate organization or individual. The remote source is often connected to the Internet.

Normally untrusted code is remote code. However, code from sources local to computer system **100** may pose a high security risk. Code from such local sources may be deemed to be untrusted code from an untrusted source. Likewise, code from a particular remote source may be considered to be reliable and to pose relatively little risk, and thus may be deemed to be trusted code from a trusted resource.

According to one embodiment of the invention, an access controller is used in conjunction with protection domains to implement security policies that allow trusted code to access more resources than untrusted code, even when the trusted and untrusted code are executed by the same principal. A security policy thus established determines what actions code executor **210** will allow the code within code stream **220** to perform. The use of typed permissions and protection domains allows policies that go beyond a simple trusted/untrusted dichotomy by allowing relatively complex permission groupings and relationships.

Protection domains, permissions and policies that may be used to establish the access rights of code shall now be described in greater detail with continued reference to FIG. **2**.

Protection Domains and Permissions

According to an embodiment of the present invention, protection domains are used to enforce security within computer systems. A protection domain can be viewed as a set of permissions granted to one or more principals. A permission is an authorization by the computer system that allows a principle to execute a particular action or function. Typically, permissions involve an authorization to perform an access to a computer resource in a particular manner. An example of an authorization is an authorization to "write" to a particular directory in a file system (e.g./home).

A permission can be represented in numerous ways in a computer system. For example, a data structure containing text instructions can represent permissions. An instruction such as "permission write/somedirectory/somefile" denotes a permission to write to file "somefile" in the directory "/somedirectory." The instruction denotes which particular action is authorized, and the computer resource upon which that particular action is authorized. In this example, the particular action authorized is to "write." The computer resources upon which the particular action is authorized is a file ("/somedirectory/somefile") in a file system of computer system **100**. Note that in the example provided the file and the directory in which the file is contained are expressed in a conventional form recognized by those skilled in the art.

Permissions can also be represented by objects, herein referred to as permission objects. Attributes of the object represent a particular permission. For example, an object can contain an action attribute of "write," and a target resource

attribute of "/somedirectory." A permission object may have one or more permission validation methods which determine whether a requested permission is authorized by the particular permission represented by the permission object.

Policies

The correlation between permissions and principals constitutes the security policy of the system. FIG. **2** illustrates an exemplary policy implemented through use of a policy file **244**. A protection domain in this exemplary policy is defined as the set of permissions granted to the objects associated with a particular code identifier. The policy of the system is represented by one or more files containing instructions. The instructions map code identifiers to authorized permissions. Each instruction establishes a mapping between a particular code identifier and a particular authorized permission. An instruction represents one authorized permission for the objects belonging to the classes associated with the code identifier in the instruction.

Storing instructions in a file is just one method of representing the policy of the system with persistently stored data. Other methods are possible for representing the policy with persistent data. For example, data in a database system can be used to map code identifiers to authorized permissions, or attributes of a persistent object can be used to map code identifiers to authorized permissions.

FIG. **3** illustrates exemplary policy file **244**. The format of an instruction in exemplary policy file **244** is:

<"permission"><URL><key name><action><target> The <URL> and <key name> represent a code identifier; the <action> and <target> represent a permission. The key name is associated with a key. The key and corresponding key name are stored together in a database. The key name can be used to find the key in the database. Instruction **320-1** in FIG. **3**, for example, is therefore an authorization of a permission to write to any file in "/tmp/*" by any object of the classes associated with code identifier "file:// somesource"-"somekey" (i.e. URL-key).

Establishing Protection Domains

In order to efficiently and conveniently implement the policy and establish protection domains, policy object **240**, domain mapper object **248**, and one or more protection domain objects **282** are provided. The policy object **240** and domain mapper **248** are initialized during the initialization of code executor **210**. A protection domain object **282** is created in a manner which shall be described. When the domain mapper is initialized, each instruction in the policy file **244** is parsed to generate a list representation of the code identifier/authorized permission combinations that together represent the policy. Each entry in the list represents a code identifier/authorized permission pair. Code identifiers and authorized permissions may each be represented with data structures or objects.

A method for establishing protection domains authorized for an object executing on computer system **200** shall now be described with reference to steps shown in FIG. **4**. The exemplary code executor **210** and objects shown in FIG. **3** will be used as example. In step **406**, a class definition is received from code stream **220** by code executor **210**. Note that the code identifier of the class is also received. The class defined by a class definition received by the code executor **210** is herein referred to as a received class. In some methods of implementing a code executor, such as a Java virtual machine, a class loader is invoked. The class loader loads code from code stream **220** that defines a class, and then executes the steps **410** to **428** in manner which shall be described.

In the present example, assume that object **262** and protection domain object **282** have not yet been created, and that code executor is receiving the class definition for the class **260** defining object **262**. In step **406**, the code identifier received for class **260** is "file://somesource"-"somekey" (i.e. URL-key). Code executor **210** then invokes a class loader.

In step **410**, a determination is made as to whether a protection domain object **282** is established for the code identifier associated with the received class. The determination is made by invoking a method of a domain mapper **248**. The method returns the protection domain object for a given code identifier, if a protection domain object has been established for the given code identifier. The domain mapper maintains data indicating which protection domain objects **282** have been created and mapping of protection domain objects **282** to the one or more codes sources that may be associated with each protection domain object.

If no protection domain object is returned, then there is no protection domain established for the code identifier. Control then passes to step **420**. In the present example, the data in the domain mapper indicating which protection domain objects **282** have been created and the code identifiers associated with each indicates there is no protection domain associated with the code identifier "file:/somesource"-"somekey" list of protection domain objects. Therefore, control passes to step **420**.

In step **420** a protection domain is established for the code identifier associated the received class. The protection domain is created by first invoking a method of policy object **240**, passing as a parameter the code identifier associated with the received class. In return, the policy object, which contains a mapping of all code identifier/authorized permissions, transmits a message having data indicating the mapping of the code identifier to the one or more authorized permissions mapped to the code identifier. The authorized permissions may be transmitted by, for example, returning a permissions container object. Then a protection domain object is created, using the permissions container object to populate the protection domain.

In the present example, the method of the policy object which returns the permissions associated with a code identifier is invoked passing the code identifier, "file:// somesource"-"somekey," as a parameter. The policy object returns a permissions container object containing all the permissions associated with the code identifier "file:// somesource" -"somekey." There is only one permission associated with the code identifier "file://somesource"-"somekey", which is a permission to write to any file in directory "Itmp/*". Then protection domain object **282** is created and populated with the permission just mentioned.

Note the policy object may determine that no protection domain is defined for a code identifier. In this case, a default protection domain is provided. Typically, a default protection domain contains permissions posing no risk to the security of computer system **100**.

Next, in step **428**, the mapping of the class to the protection domain is established. The mapping of the class to the protection domain is added to a mapping data structure maintained within the domain mapper **248**. In this example, a mapping between class **260** and protection domain object **282** is created.

Creating an association of classes to protection domains in the manner just described offers several advantages. First, because the permissions authorized to an object are based on the code identifier associated with the object's class, an object's permission can based on the source of code creating

the object. This enables authorization of permissions to be organized according to the source of code executed by code executor **210**. This ability facilitates development of more granular security mechanisms. For example, objects based on code from a first untrusted source can be granted one set of permissions and objects based on code from a second untrusted source can be granted a second set of permissions

In addition to being able to organize a policy with greater granularity, the policy can be configured with very little programming or in-depth knowledge of security systems. As described earlier, simple instructions can be entered and received by the computer system and stored in a policy file. The policy file is then used by the computer system to establish the security policy for the system.

In other embodiments of the invention, instead of storing the mapping of classes to protection domains in a domain mapper object, the mapping is stored as static fields in the protection domain class. The protection domain class is the class to which protection domain objects belong. There is only one instance of a static field for a class no matter how many objects belong to the class. The data indicating which protection domains have been created and the code identifiers associated with the protection domains is stored in static fields of the protection domain class. Alternatively, a mapping between a class and protections domains associated with the class is stored as static fields in the class.

Static methods are used to access and update the static data mentioned above. Static methods are invoked on behalf of the entire class, and may be invoked without referencing a specific object.

Exemplary Access Control

An exemplary method using access controller **280** according to steps shown in FIG. **5** will illustrate a use of objects and data structures described above. The calling stack **610** and protection domains shown in FIG. **6** are used as an example illustrating the performance of the steps shown in FIG. **5**.

A code executor, such as a Java virtual machine, maintains for each thread or process a call stack of the object methods invoked by the thread or process. The call stack reflects the calling hierarchy between the methods that have been invoked but not yet completed by the thread or process. The call stack includes information identifying the objects with methods on the call stack. For example, assume that a thread executes a.x (where "a" is an object and "x" is a method associated with object "a"). Assume that a.x invokes b.y which invokes c.z. While c.z is executing, the call stack will contain data identifying a.x, b.y, and c.z. At this point, call stack **610** represents the calling hierarchy of the methods invoked by the thread but have not yet been completed by the thread. When the thread finishes execution of c.z, the data identifying c.z will be removed from the call stack.

Note that objects corresponding to the method invocations in the call stack are each associated with a protection domain. Object a is associated with protection domain I and object b and object c are associated with protection domain J. Each protection domain object is associated with permission objects. The association between the objects, permission domain objects and the permission objects is based on the domain mapper, policy object, a policy file, and constitutes the security policy with respect to the objects shown in FIG. **6**.

Assume that a thread invokes a.x, b.y, and c.z in the manner described so that call stack **610** is as it appears in FIG. **6**. Referring to FIG. **5**, assume in step **550** a request to perform a particular action is then received from the thread while thread is executing c.z. Typically, a request is in the form of an attempt to invoke a particular method that performs a particular operation. In this example, the particular request is made by object a. In other words, a method associated with object a invoked a method that may perform the particular action. Object a is requesting to write to file "/tmp/temporary". The permission required to perform this action is a "write" permission for file "/tmp/temporary". The permission required to perform a requested action is herein referred to as a required permission.

Typically, access to a resource by code being executed by a code executor can only be made by invoking a resource manager. A resource manager is an object assigned the responsibility of managing access to its respective resource. A resource manager receives the request from object a. In response to receiving the request from object a, the resource manager assigned to manage the file system invokes an access controller. The access controller determines whether the permission required is authorized for the entity requesting access. In this example, access controller **280** is invoked by the resource manager that received the request from object c.

In step **564**, a determination is made as to whether the required permission for the requested action is included in the protection domains associated with the request to perform an action. The protection domains associated with the request are the protection domains associated with each object represented in the call stack when the request for access was made. A requested action is authorized if every protection domain associated with the objects represented by the call stack when the request for the requested action was made contains a permission authorizing the permission required to perform the requested action. Each permission object of each particular protection domain object associated with each object represented by the call stack is examined in order to determine whether the permission object authorizes the required permission.

Examining the permissions of a particular protection domain associated with an object begins by determining an object's class. A code executor, such as a Java virtual machine, provides that each object incorporates a method which returns the class of an object. In this example, the first object with a method on the call stack is object a. Access controller **280** invokes the method that returns the class of object a.

Next, the method of the class/domain mapper that returns the protection domain object associated with a class is invoked. Each permission in the returned protection domain object is examined until it is determined whether any permission in the protection domain authorizes the required permission.

The validation method of each permission object in the returned protection domain object is invoked until the validation method of a permission object indicates that the required permission is authorized. As mentioned earlier, a permission object contains a permission validation method which indicates whether a particular permission is authorized by the permission represented by the permission object. When a protection domain is encountered that does not contain a permission authorizing the required permission, then execution of the steps ceases. If every permission object contained by the protection domain authorizes the required permission, then the next protection domain for the next object represented on the calling stack is examined, if any. After the last protection domain is

examined and found to contain the required permission, control passes to step **568**, where the action is performed.

In this example, protection domain object I is returned as the protection domain associated with the class of object a. The first permission object in protection domain object a represents an authorized permission to write to any file in directory "/tmp". When the validation method of the first permission object is invoked the validation method indicates that the required permission is authorized.

Access controller **280** then examines the protection domain for the class of the next object represented in the call stack, which is b. Protection domain object J is the protection domain associated with the class of object b. The first permission object in protection domain object J represents an authorized permission to write to any file in directory "/share". When the validation method of the first permission object is invoked the validation method indicates that the required permission is authorized. The next permission object is examined. When the validation method of the second permission object is invoked the validation method indicates that the required permission is not authorized. Thus execution of the steps ceases **560**.

In the foregoing specification, the invention has been described with reference to specific embodiments thereof. It will, however, be evident that various modifications and changes may be made thereto without departing from the broader spirit and scope of the invention. The specification and drawings are, accordingly, to be regarded in an illustrative rather than a restrictive sense.

What is claimed is:

1. A method for providing security, the method comprising the steps of:

establishing one or more protection domains, wherein a protection domain is associated with zero or more permissions;

establishing an association between said one or more protection domains and one or more classes of one or more objects; and

determining whether an action requested by a particular object is permitted based on said association between said one or more protection domains and said one or more classes.

2. The method of claim **1**, wherein:

at least one protection domain of said one or more protection domains is associated with a code identifier;

at least one class of said one or more classes is associated with said code identifier; and

the step of establishing an association between said one or more protection domains and said one or more classes of one or more objects further includes the step of associating said one or more protection domains and said one or more classes based on said code identifier.

3. The method of claim **2**, wherein said code identifier indicates a source of code used to define each class of said one or more classes.

4. The method of claim **2**, wherein said code identifier indicates a key associated with each class of said one or more classes.

5. The method of claim **2**, wherein said code identifier indicates a source of code used to define each class of said one or more classes and indicates a key associated with each class of said one or more classes.

6. The method of claim **2**, wherein the step of associating said one or more protection domains and said one or more classes based on said code identifier further includes associating said one or more protection domains and said one or more classes based on data persistently stored, wherein said data associates code identifiers with a set of one or more permissions.

7. A method of providing security, the method comprising the steps of:

establishing one or more protection domains, wherein a protection domain is associated with zero or more permissions;

establishing an association between said one or more protection domains and one or more sources of code; and

in response to executing code making a request to perform an action, determining whether said request is permitted based on a source of said code making said request and said association between said one or more protection domains and said one or more sources of code.

8. The method of claim **7**, wherein the step of establishing an association between said one or more protection domains and said one or more sources of code further includes establishing an association between said one or more protection domains and said one or more sources of code and one or more keys associated with said one or more sources of code.

9. The method of claim **8**, wherein the step of establishing an association between said one or more protection domains and said one or more sources of code and said one or more keys associated with said one or more sources of code further includes establishing said association between said one or more protection domains and said one or more sources of code and said one or more keys associated with said one or more sources of code based on data persistently stored, wherein said data associates particular sources of code and particular keys with a set of one or more permissions.

10. A computer-readable medium carrying one or more sequences of one or more instructions, the one or more sequences of the one or more instructions including instructions which, when executed by one or more processors, causes the one or more processors to perform the steps of:

establishing one or more protection domains, wherein a protection domain is associated with zero or more permissions;

establishing an association between said one or more protection domains and one or more classes of one or more objects; and

determining whether an action requested by a particular object is permitted based on said association between said one or more protection domains and said one or more classes.

11. The computer readable medium of claim **10**, wherein:

at least one protection domain of said one or more protection domains is associated with a code identifier;

at least one class of said one or more classes is associated with said code identifier; and

the step of establishing an association between said one or more protection domains and said one or more classes of one or more objects further includes the step of associating said one or more protection domains and said one or more classes based on said code identifier.

12. The computer readable medium of claim **11**, wherein said code identifier indicates a source of code used to define each class of said one or more classes.

13. The computer readable medium of claim **11**, wherein said code identifier indicates a key associated with each class of said one or more classes.

14. The computer readable medium of claim **11**, wherein said code identifier indicates a source of code used to define

each class of said one or more classes and indicates a key associated with each class of said one or more classes.

15. The computer readable medium of claim **14**, wherein the step of associating said one or more protection domains and said one or more classes based on said code identifier further includes associating said one or more protection domains and said one or more classes based on data persistently stored, wherein said data associates code identifiers with a set of one or more permissions.

16. A computer-readable medium carrying one or more sequences of one or more instructions, wherein the execution of the one or more sequences of the one or more instructions causes the one or more processors to perform the steps of:

establishing one or more protection domains, wherein a protection domain is associated with zero or more permissions;

establishing an association between said one or more protection domains and one or more sources of code; and

in response to executing code making a request to perform an action, determining whether said request is permitted based on a source of said code making said request and said association between said one or more protection domains and said one or more sources of code.

17. The computer readable medium of claim **16**, wherein the step of establishing an association between said one or more protection domains and said one or more sources of code further includes establishing an association between said one or more protection domains and said one or more sources of code and one or more keys associated with said one or more sources of code.

18. The computer readable medium of claim **17**, wherein the step of establishing an association between said one or more protection domains and said one or more sources of code and said one or more keys associated with said one or more sources of code further includes establishing said association between said one or more protection domains and said one or more sources of code and said one or more keys associated with said one or more sources of code based on data persistently stored, wherein said data associates particular sources of code and particular keys with a set of one or more permissions.

19. A computer system comprising:

a processor;

a memory coupled to said processor;

one or more protection domains stored as objects in said memory, wherein each protection domain is associated with zero or more permissions;

a domain mapping object stored in said memory, said domain mapping object establishing an association between said one or more protection domains and one or more classes of one or more objects; and

said processor being configured to determine whether an action requested by a particular object is permitted based on said association between said one or more protection domains and said one or more classes.

20. The computer system of claim **19**, wherein:

at least one protection domain of said one or more protection domains is associated with a code identifier;

at least one class of said one or more classes is associated with said code identifier; and

said computer system further comprises said processor configured to establish an association between said one or more protection domains and said one or more classes of one or more objects by associating said one or more protection domains and said one or more classes based on said code identifier.

21. The computer system of claim **20**, wherein said code identifier indicates a source of code used to define each class of said one or more classes.

22. The computer system of claim **20**, wherein said code identifier indicates a key associated with each class of said one or more classes.

23. The computer system of claim **20**, wherein said code identifier indicates a source of code used to define each class of said one or more classes and indicates a key associated with each class of said one or more classes.

24. The computer system of claim **20**, further comprising said processor configured to associate said one or more protection domains and said one or more classes based on said code identifier by associating said one or more protection domains and said one or more classes based on data persistently stored in said computer system, wherein said data associates code identifiers with a set of one or more permissions.

* * * * *

EXHIBIT B

US006192476B1

(12) **United States Patent**
Gong

(10) **Patent No.: US 6,192,476 B1**
(45) **Date of Patent: Feb. 20, 2001**

(54) **CONTROLLING ACCESS TO A RESOURCE**

(75) Inventor: **Li Gong**, Menlo Park, CA (US)

(73) Assignee: **Sun Microsystems, Inc.**, Mountain View, CA (US)

(*) Notice: Under 35 U.S.C. 154(b), the term of this patent shall be extended for 0 days.

(21) Appl. No.: **08/988,431**

(22) Filed: **Dec. 11, 1997**

(51) **Int. Cl.**[7] **H04L 9/00**
(52) **U.S. Cl.** **713/201**; 713/152; 709/229
(58) **Field of Search** 713/200, 201–202, 713/152–153, 154, 117, 187, 188; 308/4; 714/38, 48; 395/704; 709/229

(56) **References Cited**

U.S. PATENT DOCUMENTS

4,809,160	*	2/1989	Mahon et al.	364/200
5,311,591		5/1994	Fischer	380/4
5,649,099		7/1997	Theimer et al.	713/201
5,720,033	*	2/1998	Deo	713/200
5,745,678	*	4/1998	Herzberg et al.	713/200
5,758,153	*	5/1998	Atsatt et al.	395/614
5,845,129	*	12/1998	Wendorf et al.	395/726
5,892,904	*	4/1999	Atkinson et al.	713/201
5,915,085	*	6/1999	Koved	713/200
5,987,608	*	11/1999	Roskind	713/200

FOREIGN PATENT DOCUMENTS

2259590A	3/1993	(GB)	G06F/9/44
2308688A	7/1997	(GB)	G06F/12/14

OTHER PUBLICATIONS

Dean, D., et al., "Java Security: From HotJava to Netscape and Beyond," Proceedings of the 1996 IEEE Symposium on Security and Privacy, Oakland, CA, May 6–8, 1996.

Hamilton, M.A., "Java and the Shift to Net–Centric Computing," Computer, vol. 29, No. 8, Aug., 1996.

Gong Li, et al.: "Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java™ Development Kit 1.2", Proceedings Of The Usenix Symposium On Internet Technologies And Systems, Monterey, CA, USA, Dec. 8–11, 1997,ISBN 1–880446–91–S, 1997, Berkeley, CA, USA, Usenix Assoc., USA, pp. 103–112, XP–002100907.

Wallach, D. S., et al.: "Extensible Security Architectures for Java", 16th ACM Symposium On Operating Systems Principles, Sain Malo, France, Oct. 5–8 1997, ISSN 0163–5980, Operating Systems Review, Dec. 1997, ACM, USA, pp. 116–128, XP–002101681.

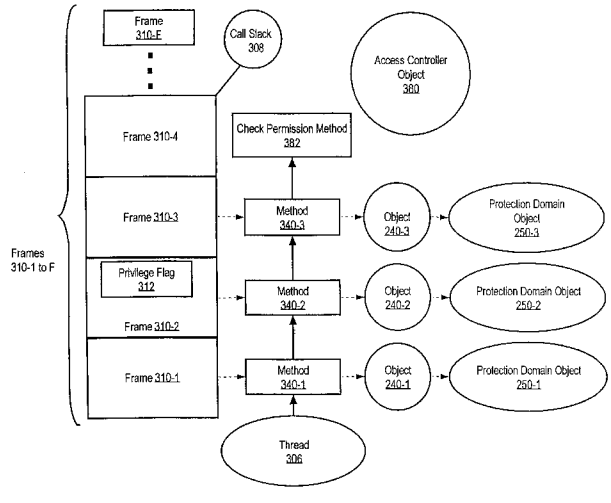
* cited by examiner

Primary Examiner—Robert W. Beausoliel, Jr.
Assistant Examiner—Scott T. Baderman
(74) *Attorney, Agent, or Firm*—McDermott, Will & Emery

(57) **ABSTRACT**

A method and system are provided for determining whether a principal (e.g. a thread) may access a particular resource. According to one aspect of the invention, the access authorization determination takes into account the sources of the code on the call stack of the principal at the time the access is desired. Because the source of the code on the call stack will vary over time, so will the access rights of the principal. Thus, when a request for an action is made by a thread, a determination is made of whether the action is authorized based on permissions associated with routines in a calling hierarchy associated with the thread. The determination of whether a request is authorized is based on a determination of whether at least one permission associated with each routine encompasses the permission required to perform the requested action. Support for "privileged" routines is also provided. When a routine in the calling hierarchy is privileged, the determination of whether an action is authorized is made by determining whether at least one permission associated with each routine between and including the privileged routine and a second routine in the calling hierarchy encompasses the permission required to perform the requested action.

21 Claims, 4 Drawing Sheets



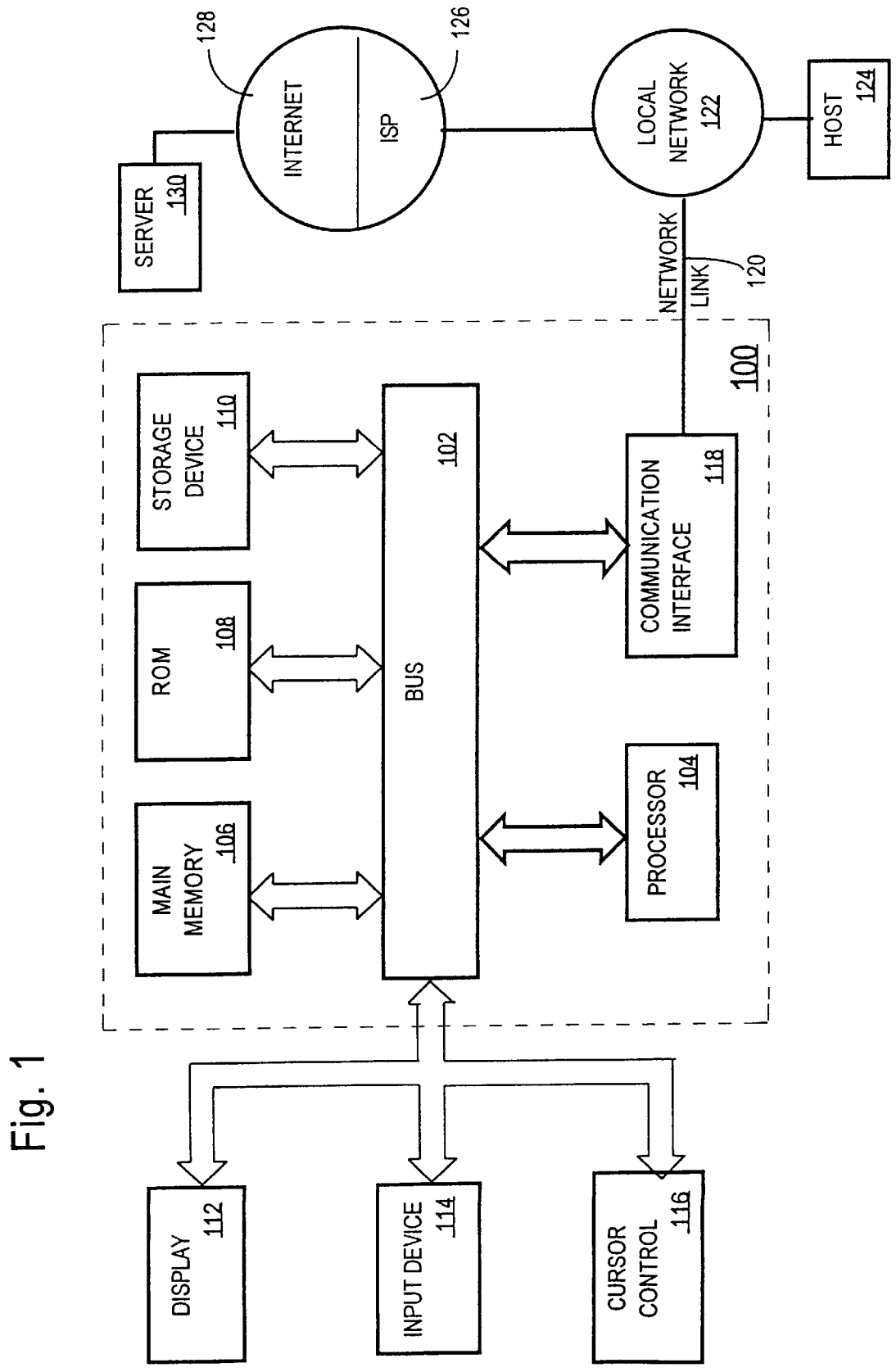

128
INTERNET
ISP
126
LOCAL NETWORK 122
HOST 124
SERVER 130
NETWORK LINK 120
100
STORAGE DEVICE 110
ROM 108
MAIN MEMORY 106
BUS 102
COMMUNICATION INTERFACE 118
PROCESSOR 104
DISPLAY 112
INPUT DEVICE 114
CURSOR CONTROL 116

Fig. 1

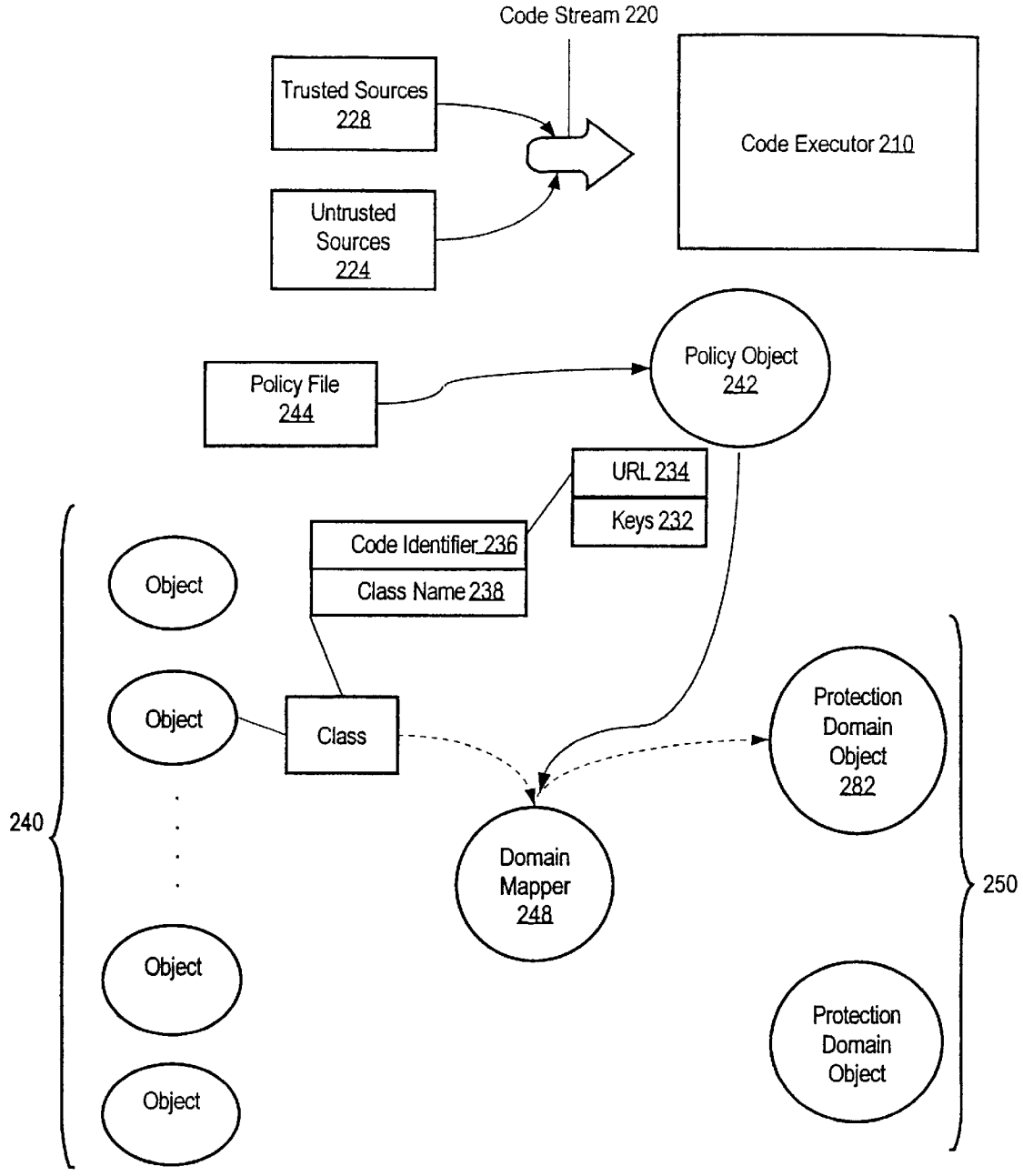

Code Stream 220
Trusted Sources 228
Untrusted Sources 224
Code Executor 210
Policy Object 242
Policy File 244
URL 234
Keys 232
Code Identifier 236
Class Name 238
Object
Object
Class
Object
Object
240
Protection Domain Object 282
Domain Mapper 248
Protection Domain Object
250

Fig. 2

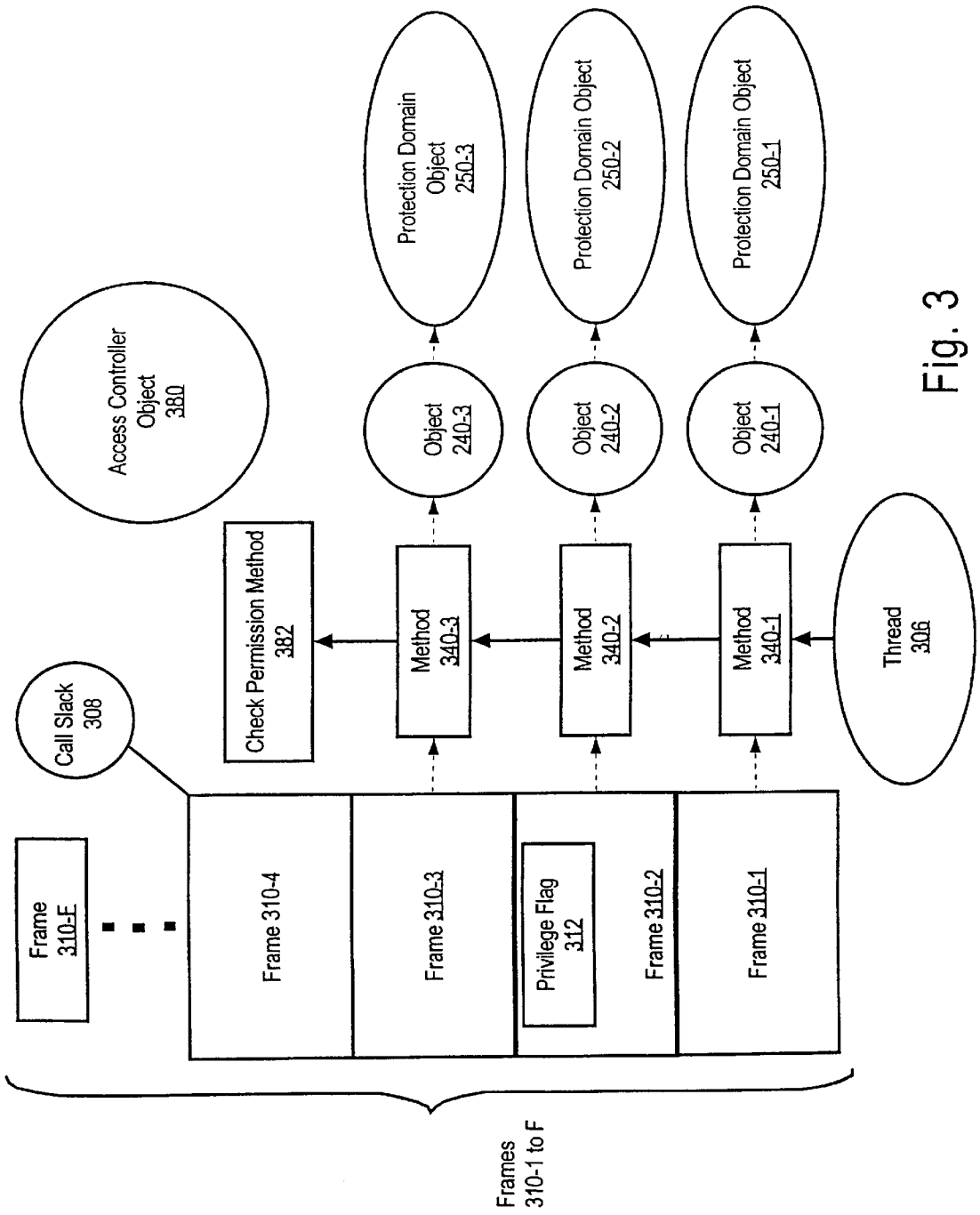

Protection Domain Object 250-3
Protection Domain Object 250-2
Protection Domain Object 250-1
Access Controller Object 380
Object 240-3
Object 240-2
Object 240-1
Check Permission Method 382
Method 340-3
Method 340-2
Method 340-1
Thread 306
Call Slack 308
Frame 310-F
Frame 310-4
Frame 310-3
Privilege Flag 312
Frame 310-2
Frame 310-1
Frames 310-1 to F

Fig. 3

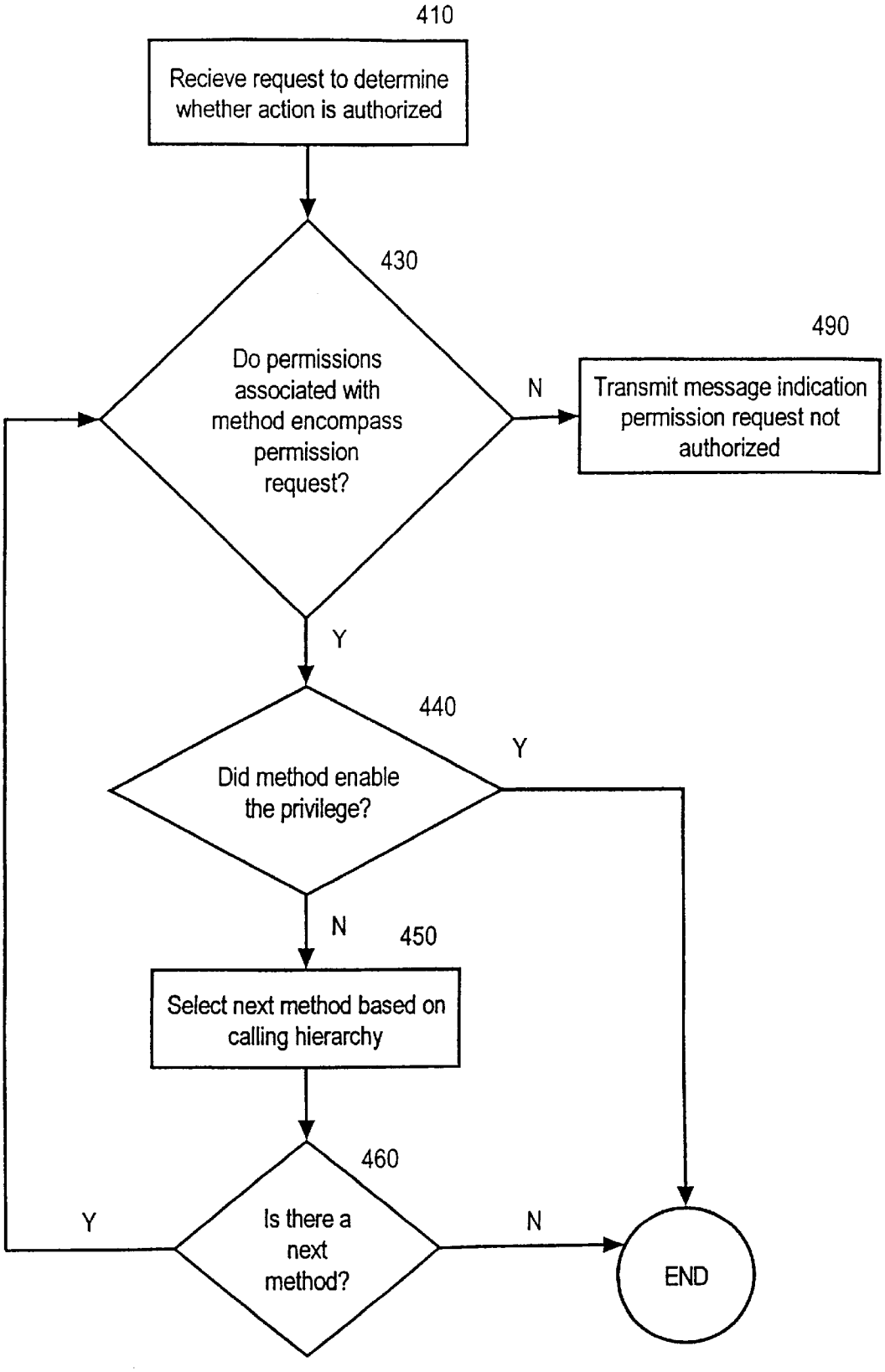

410
Recieve request to determine whether action is authorized
430
Do permissions associated with method encompass permission request?
N
490
Transmit message indication permission request not authorized
Y
440
Did method enable the privilege?
Y
N
450
Select next method based on calling hierarchy
460
Is there a next method?
Y
N
END

Fig. 4

CONTROLLING ACCESS TO A RESOURCE

RELATED APPLICATIONS

The present application is related to U.S. patent application Ser. No. 08/988,857, entitled "TYPED, PARAMETERIZED, AND EXTENSIBLE ACCESS CONTROL PERMISSIONS", filed by Li Gong, on the equal day herewith, the contents of which are incorporated herein by reference.

The present application is related to U.S. patent application Ser. No. 08/988,660, entitled "SECURE CLASS RESOLUTION, LOADING, AND DEFINITION", filed by Li Gong, on the equal day herewith, the contents of which are incorporated herein by reference.

The present application is related to U.S. patent application Ser. No. 08/988,439, entitled "PROTECTION DOMAINS TO PROVIDE SECURITY IN A COMPUTER SYSTEM", filed by Li Gong, on the equal day herewith, the contents of which are incorporated herein by reference.

FIELD OF THE INVENTION

The present invention relates to security mechanisms in a computer system.

BACKGROUND OF THE INVENTION

As the use of computer systems grows, organizations are becoming increasingly reliant upon them. A malfunction in the computer system can severely hamper the operation of such organizations. Thus organizations that use computer systems are vulnerable to users who may intentionally or unintentionally cause the computer system to malfunction.

One way to compromise the security of a computer system is to cause the computer system to execute software that performs harmful actions on the computer system. There are various types of security measures that may be used to prevent a computer system from executing harmful software. One example is to check all software executed by the computer system with a "virus" checker. However, virus checkers only search for very specific software instructions. Many methods of using software to tamper with a computer's resources would not be detected by a virus checker.

Another very common measure used to prevent the execution of software that tampers with a computer's resources is the "trusted developers approach". According to the trusted developers approach, system administrators limit the software that a computer system can access to only software developed by trusted software developers. Such trusted developers may include, for example, well know vendors or in-house developers.

Fundamental to the trusted developers approach is the idea that computer programs are created by developers, and that some developers can be trusted to not have produced software that compromises security. Also fundamental to the trusted developers approach is the notion that a computer system will only execute programs that are stored at locations that are under control of the system administrators.

Recently developed methods of running applications involve the automatic and immediate execution of software code loaded from remote sources over a network. When the network includes remote sources that are outside the control of system administrators, the trusted developers approach does not work.

One attempt to adapt the trusted developers approach to systems that can execute code from remote sources is

referred to as the trusted source approach. Key to the trusted source approach is the notion that the location from which a program is received (e.g. the "source" of the program) identifies the developer of the program. Consequently, the source of the program may be used to determine whether the program is from a trusted developer. If the source is associated with a trusted developer, then the source is considered to be a "trusted source" and execution of the code is allowed.

One implementation of the trusted source approach is referred to as the sand box method. The sand box method allows all code to be executed, but places restrictions on remote code. Specifically, the sand box method permits all trusted code full access to a computer system's resources and all remote code limited access to a computer system's resources. Trusted code is usually stored locally on the computer system under the direct control of the owners or administrators of the computer system, who are accountable for the security of the trusted code.

One drawback to the sandbox approach is that the approach is not very granular. The sandbox approach is not very granular because all remote code is restricted to the same limited set of resources. Very often, there is a need to permit remote code from one source access to one set of computer resources while permitting remote code from another source access to another set of computer resources. For example, there may be a need to limit access to one set of files associated with one bank to remote code loaded over a network from a source associated with that one bank, and limit access to another set of files associated with another bank to remote code loaded over a network from a source associated with the other bank.

Providing security measures that allow more granularity than the sand box method involves establishing a complex set of relationships between principals and permissions. A "principal" is an entity in the computer system to which permissions are granted. Examples of principals include processes, objects and threads. A "permission" is an authorization by the computer system that allows a principal to perform a particular action or function.

The task of assigning permissions to principals is complicated by the fact that sophisticated processes may involve the interaction of code from multiple sources. For example, code from a trusted first source being executed by a principal (c.g. thread) may cause the execution of code from a trusted second source, and then cause execution of code from an untrusted third source. Even though the principal remains the same when the code from the trusted second source and code from the untrusted third source is being executed, the access privileges appropriate for the principal when code from the trusted second source is being executed likely differ from access privileges appropriate for the principal when the code from the untrusted third source is being executed. Thus, access privileges appropriate for a principal may change dynamically as the source of the code being executed by the principal changes.

Based on the foregoing, it is clearly desirable to develop a security method which can determine the appropriate access privileges for principals. It is further desirable to provide a security method that allows permissions to change dynamically when code from one source causes the execution of code from another source.

SUMMARY OF THE INVENTION

A method and apparatus for determining the access rights of principals is provided. According to one aspect of the invention, access rights for a principal are determined

dynamically based on the source of the code that is currently being executed by the principal (e.g. thread, process)

According to one aspect of the invention, when a request for an action by a thread is detected, a determination is made of whether the action is authorized based on permissions associated with routines in a calling hierarchy associated with the thread. A calling hierarchy indicates the routines (e.g. functions, methods) that have been invoked by or on behalf of a principal (e.g. thread, process) but have not been exited.

In one embodiment, the association between permissions and routines is based on an association between routines and classes and between classes and protection domains. Thus, for example, a first routine may be associated with a first set of permissions, which correspond to the permissions belonging to the protection domain associated with the code source of the first routine's associated class. A second routine may be associated with a second set of permissions, which correspond to the permissions belonging to the protection domain associated with the code source of the second routine's associated class. The determination of whether a particular request is authorized is based on a determination of whether at least one permission associated with each routine in the calling hierarchy encompasses the permission required to perform the requested action. For example, a determination of whether a particular request is authorized is based on determining whether (1) at least one permission from a first set of permissions associated with the first routine in a calling hierarchy encompass the permission required, and (2) at least one permission of a second set of permissions associated with the second routine in the calling hierarchy encompass the permission required.

According to another aspect of the invention, certain routines may be "privileged". A privileged routine is allowed to perform certain actions even if the routine that called the privileged routine does not have permission to perform those same actions.

According to one embodiment, a flag in a frame in the calling hierarchy corresponding to a privileged routine is set to indicate that the privileged routine is privileged. A frame is a data element in a calling hierarchy that corresponds to an invocation of a routine (e.g. function, method) that has not been exited. When a first routine in the calling hierarchy is privileged, the determination of whether an action is authorized is made by determining whether at least one permission associated with each routine between and including the first routine and a second routine in the calling hierarchy encompasses the permission required to perform the requested action. The permissions of the routines preceding the privileged routine in the calling hierarchy are ignored.

BRIEF DESCRIPTION OF THE DRAWINGS

The present invention is illustrated by way of example, and not by way of limitation, in the figures of the accompanying drawings and in which like reference numerals refer to similar elements and in which:

FIG. **1** is a block diagram of a computer system on which the present invention may be implemented in accordance with an embodiment of the present invention;

FIG. **2** is a block diagram showing exemplary protection domain objects, a code executor, classes, and objects created by the code executor in accordance with an embodiment of the present invention;

FIG. **3** is a block diagram illustrating an exemplary access controller, call stack, protection domain, and the object and methods being executed by a thread sending a request to determine whether an action is authorized to the access controller in accordance with an embodiment of the present invention; and

FIG. **4** is a flow chart showing the steps of a method used to determine whether an action is authorized for a thread in accordance with an embodiment of the present invention.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

A method and apparatus for determining authorization to perform actions on a computer system is described. In the following description, for the purposes of explanation, numerous specific details are set forth in order to provide a thorough understanding of the present invention. It will be apparent, however, to one skilled in the art that the present invention may be practiced without these specific details. In other instances, well-known structures and devices are shown in block diagram form in order to avoid unnecessarily obscuring the present invention.

HARDWARE OVERVIEW

FIG. **1** is a block diagram that illustrates a computer system **100** upon which an embodiment of the invention may be implemented. Computer system **100** includes a bus **102** or other communication mechanism for communicating information, and a processor **104** coupled with bus **102** for processing information. Computer system **100** also includes a main memory **106**, such as a random access memory (RAM) or other dynamic storage device, coupled to bus **102** for storing information and instructions to be executed by processor **104**. Main memory **106** also may be used for storing temporary variables or other intermediate information during execution of instructions to be executed by processor **104**. Computer system **100** further includes a read only memory (ROM) **108** or other static storage device coupled to bus **102** for storing static information and instructions for processor **104**. A storage device **110**, such as a magnetic disk or optical disk, is provided and coupled to bus **102** for storing information and instructions.

Computer system **100** may be coupled via bus **102** to a display **112**, such as a cathode ray tube (CRT), for displaying information to a computer user. An input device **114**, including alphanumeric and other keys, is coupled to bus **102** for communicating information and command selections to processor **104**. Another type of user input device is cursor control **116**, such as a mouse, a trackball, or cursor direction keys for communicating direction information and command selections to processor **104** and for controlling cursor movement on display **112**. This input device typically has two degrees of freedom in two axes, a first axis (e.g., x) and a second axis (e.g., y), that allows the device to specify positions in a plane.

The invention is related to the use of computer system **100** for determining authorization to perform actions on a computer system. According to one embodiment of the invention, determining authorization to perform actions on a computer system is provided by computer system **100** in response to processor **104** executing one or more sequences of one or more instructions contained in main memory **106**. Such instructions may be read into main memory **106** from another computer-readable medium, such as storage device **110**. Execution of the sequences of instructions contained in main memory **106** causes processor **104** to perform the process steps described herein. In alternative embodiments, hard-wired circuitry may be used in place of or in combi-

nation with software instructions to implement the invention. Thus, embodiments of the invention are not limited to any specific combination of hardware circuitry and software.

The term "computer-readable medium" as used herein refers to any medium that participates in providing instructions to processor **104** for execution. Such a medium may take many forms, including but not limited to, non-volatile media, volatile media, and transmission media. Non-volatile media includes, for example, optical or magnetic disks, such as storage device **110**. Volatile media includes dynamic memory, such as main memory **106**. Transmission media includes coaxial cables, copper wire and fiber optics, including the wires that comprise bus **102**. Transmission media can also take the form of acoustic or light waves, such as those generated during radio-wave and infra-red data communications.

Common forms of computer-readable media include, for example, a floppy disk, a flexible disk, hard disk, magnetic tape, or any other magnetic medium, a CD-ROM, any other optical medium, punchcards, papertape, any other physical medium with patterns of holes, a RAM, a PROM, and EPROM, a FLASH-EPROM, any other memory chip or cartridge, a carrier wave as described hereinafter, or any other medium from which a computer can read.

Various forms of computer readable media may be involved in carrying one or more sequences of one or more instructions to processor **104** for execution. For example, the instructions may initially be carried on a magnetic disk of a remote computer. The remote computer can load the instructions into its dynamic memory and send the instructions over a telephone line using a modem. A modem local to computer system **100** can receive the data on the telephone line and use an infra-red transmitter to convert the data to an infra-red signal. An infra-red detector coupled to bus **102** can receive the data carried in the infra-red signal and place the data on bus **102**. Bus **102** carries the data to main memory **106**, from which processor **104** retrieves and executes the instructions. The instructions received by main memory **106** may optionally be stored on storage device **110** either before or after execution by processor **104**.

Computer system **100** also includes a communication interface **118** coupled to bus **102**. Communication interface **118** provides a two-way data communication coupling to a network link **120** that is connected to a local network **122**. For example, communication interface **118** may be an integrated services digital network (ISDN) card or a modem to provide a data communication connection to a corresponding type of telephone line. As another example, communication interface **118** may be a local area network (LAN) card to provide a data communication connection to a compatible LAN. Wireless links may also be implemented. In any such implementation, communication interface **118** sends and receives electrical, electromagnetic or optical signals that carry digital data streams representing various types of information.

Network link **120** typically provides data communication through one or more networks to other data devices. For example, network link **120** may provide a connection through local network **122** to a host computer **124** or to data equipment operated by an Internet Service Provider (ISP) **126**. ISP **126** in turn provides data communication services through the world wide packet data communication network now commonly referred to as the "Internet" **128**. Local network **122** and Internet **128** both use electrical, electromagnetic or optical signals that carry digital data streams. The signals through the various networks and the signals on network link **120** and through communication interface **118**, which carry the digital data to and from computer system **100**, are exemplary forms of carrier waves transporting the information.

Computer system **100** can send messages and receive data, including program code, through the network(s), network link **120** and communication interface **118**. In the Internet example, a server **130** might transmit a requested code for an application program through Internet **128**, ISP **126**, local network **122** and communication interface **118**. In accordance with the invention, one such downloaded application provides for determining authorization to perform actions on a computer system as described herein.

The received code may be executed by processor **104** as it is received, and/or stored in storage device **110**, or other non-volatile storage for later execution. In this manner, computer system **100** may obtain application code in the form of a carrier wave.

FUNCTIONAL OVERVIEW

A security enforcement mechanism is provided in which the access permissions of a principal, such as a thread, are allowed to vary over time based on the source of the code currently being executed. When a routine that arrives from a trusted source is executing, the thread executing the routine is typically allowed greater access to resources. When the same thread is executing a routine from an untrusted source, the thread is typically allowed more restricted access to resources. When a routine calls another routine, the thread executing the routines is associated with permissions common to both routines, and is thus is restricted to a level of access that is lesser than or equal to the level access allowed for either routine.

The mechanism allows certain routines to be "privileged". When determining whether a thread is able to perform an action, only the permissions associated with the privileged routine and the routines above the privileged routine in the calling hierarchy of the thread are inspected.

According to one embodiment, the security mechanism described herein uses permission objects and protection domain objects to store information that models the security policy of a system. The nature and use of these objects, as well as the techniques for dynamically determining the time-variant access privileges of a principal, are described hereafter in greater detail.

EXEMPLARY SECURITY MECHANISM

An exemplary security mechanism illustrating the use of protection domains is shown in FIG. **2**. Referring to FIG. **2**, the exemplary security mechanism includes a policy file **244**, a policy object **240**, a domain mapper object **248**, an access controller **280**, and one or more protection domains **282**. The security mechanism is implemented using a code executor **210**.

Code executor **210** executes code which code executor **210** receives from code stream **220**. One example of a code executor is a Java virtual machine. A Java virtual machine interprets code called byte code. Byte code is code generated by a Java compiler from source files containing text. The Java virtual machine is described in detail in Tim Lindholm & Frank Yellin, *The Java Virtual Machine Specification* (1996).

For the purposes of explanation, it shall be assumed that code from code stream **220** is object oriented software. Consequently, the code is in the form of methods associated

with objects that belong to classes. In response to instructions embodied by code executed by code executor **210**, code executor **210** creates one or more objects **240**. An object is a record of data combined with the procedures and functions that manipulate the record. All objects belong to a class. Each object belonging to a class has the same fields ("attributes") and the same methods. The methods are the procedures, functions, or routines used to manipulate the object. An object is said to be an "instance" of the class to which the object belongs.

One or more class definitions are contained in code from code stream **220**. The fields and methods of the objects belonging to a class are defined by a class definition. These class definitions are used by code executor **210** to create objects which are instances of the classes defined by the class definitions.

These class definitions are generated from source code written by a programmer. For example, a programmer using a Java Development Kit enters source code that conforms to the Java programming language into a source file. The source code embodies class definitions and other instructions which are used to generate byte code which controls the execution of a code executor (i.e. a Java virtual machine). Techniques for defining classes and generating code executed by a code executor, such as a Java virtual machine, are well known to those skilled in the art.

Each class defined by a class definition from code stream **220** is associated with a class name **238** and a code source **236**. Code executor **210** maintains an association between a class and its class name and code source. The code source represents a source of code. A "source of code" is an entity from which computer instructions are received. Examples of sources of code include a file or persistent object stored on a data server connected over a network, a FLASH_EPROM reader that reads instructions stored on a FLASH_EPROM, or a set of system libraries.

In one embodiment of the present invention, the code source may be a composite record containing a uniform resource locator ("URL") **234** and set of public cryptographic keys **236**. A URL identifies a particular source. The URL is a string used to uniquely identify any server connected to the world wide web. A URL may also be used to designate sources local to computer system **100**. Typically, the URL includes the designation of the file and directory of the file that is the source of the code stream that a server is providing.

A public cryptographic key, herein referred to as a key, is used to validate the digital signature which may be included in a file used to transport related code and data. Public cryptographic keys and digital signatures are described in Schneier, *Applied Cryptography*, (1996). The keys may be contained in the file, may be contained in a database associating keys with sources (e.g. URLs), or be accessible using other possible alternative techniques.

A class may be associated with the digital signature associated with the file used to transport code defining the class, or the class definition of the class may be specifically associated with a digital signature. A class that is associated with a valid digital signature is referred to as being signed. Valid digital signatures are digital signatures that can be verified by known keys stored in a database. If a class is associated with a digital signature which can not be verified, or the class is not associated with any digital signature, the class is referred to as being unsigned. Unsigned classes may be associated with a default key. A key may be associated with a name, which may be used to look up the key in the database.

While one code source format has been described as including data indicating a cryptographic key and URL, alternate formats are possible. Other information indicating the source of the code, or combinations thereof, may be used to represent code sources. Therefore, it is understood that the present invention, is not limited to any particular format for a code source.

TRUSTED AND UNTRUSTED SOURCES

The source of code stream **220** may be from zero or more untrusted sources **224** or zero or more trusted sources **228**. Untrusted sources **224** and trusted sources **228** may be file servers, including file servers that are part of the World Wide Web network of servers connected to the Internet. An untrusted source is typically not under the direct control of the operators of computer system **100**. Code from untrusted sources is herein referred to as untrusted code.

Because untrusted code is considered to pose a high security risk, the set of computer resources that untrusted code may access is usually restricted to those which do not pose security threats. Code from a trusted source is code usually developed by trusted developers. Trusted code is considered to be reliable and pose much less security risk than remote code.

Software code which is loaded over the network from a remote source and immediately executed is herein referred to as remote code. Typically, a remote source is a computer system of another separate organization or individual. The remote source is often connected to the Internet.

Normally untrusted code is remote code. However, code from sources local to computer system **100** may pose a high security risk. Code from such local sources may be deemed to be untrusted code from an untrusted source. Likewise, code from a particular remote source may be considered to be reliable and to pose relatively little risk, and thus may be deemed to be trusted code from a trusted resource.

According to one embodiment of the invention, an access controller is used in conjunction with protection domains to implement security policies that allow trusted code to access more resources than untrusted code, even when the trusted and untrusted code are executed by the same principal. A security policy thus established determines what actions code executor **210** will allow the code within code stream **220** to perform. The use of typed permissions and protection domains allows policies that go beyond a simple trusted/untrusted dichotomy by allowing relatively complex permission groupings and relationships.

Protection domains and policies that may be used in conjunction with typed permissions shall now be described in greater detail with continued reference to FIG. **2**.

PROTECTION DOMAINS AND PERMISSIONS

According to an embodiment of the present invention, protection domains are used to enforce security within computer systems. A protection domain can be viewed as a set of permissions granted to one or more principals. A permission is an authorization by the computer system that allows a principal to execute a particular action or function. Typically, permissions involve an authorization to perform an access to a computer resource in a particular manner. An example of an authorization is an authorization to "write" to a particular directory in a file system (e.g. /home).

A permission can be represented in numerous ways in a computer system. For example, a data structure containing text instructions can represent permissions. An instruction

such as "permission write /somedirectory/somefile" denotes a permission to write to file "somefile" in the directory "/somedirectory." The instruction denotes which particular action is authorized, and the computer resource upon which that particular action is authorized. In this example, the particular action authorized is to "write." The computer resources upon which the particular action is authorized is a file ("/somedirectory/somefile") in a file system of computer system **100**. Note that in the example provided the file and the directory in which the file is contained are expressed in a conventional form recognized by those skilled in the art.

Permissions can also be represented by objects, herein referred to as permission objects. Attributes of the object represent a particular permission. For example, an object can contain an action attribute of "write," and a target resource attribute of "/somedirectory." A permission object may have one or more permission validation methods which determine whether a requested permission is authorized by the particular permission represented by the permission object.

POLICIES

The correlation between permissions and principals constitutes the security policy of the system. The policy of the system may be represented by one or more files containing instructions. Each instruction establishes a mapping between a particular code identifier and a particular authorized permission. The permission identified in an instruction applies to all objects that belong to the classes that are associated with the code identifier identified in the instruction.

FIG. **2** illustrates an exemplary policy implemented through use of a policy file **244**. The format of an instruction in exemplary policy file **244** is:

<"permission"> <URL> <key name> <action> <target>

The combination of the <URL> and the key that corresponds to <key name> constitute a code identifier; the <action> and <target> represent a permission. A key is associated with a key name. The key and the corresponding key name are stored together in a key database. The key name can be used to find the key in the key database. For example, consider the following instruction:

File://somesource some key write /tmp/* The above instruction represents an authorization of a permission to write to any file in "/tmp/*" by any object that belongs to the class associated with code identifier "file://somesource" -"some key" (i.e. URL-key name).

IMPLIED PERMISSIONS

One permission does not have to exactly match another permission to be considered "encompassed" by the other permission. When a first permission encompasses a second permission without matching the second permission, the first permission is said to "imply" the second permission. For example, a permission to write to any file in a directory, such "c:/", implies a permission to write to any specific file in the directory, such as "c:/thisfile".

If a permission is represented by a permission object, the validation method for the permission object contains code for determining whether one permission is implied by another. For purposes of illustration, a permission to write to any file in a directory implies a permission to write to any specific file in that directory, and a permission to read from any file in a directory implies a permission to read from any specific file in that directory, however, a permission to write does not imply a permission to read.

POLICY IMPLEMENTING OBJECTS

A variety of objects may be used to implement the policy represented by the code identifiers to permissions mapping contained in policy file **244**. According to the embodiment illustrated in FIG. **2**, in order to efficiently and conveniently implement the policy, policy object **242**, domain mapper object **248**, and one or more protection domain objects **250** are provided.

Policy object **242** is an object for storing the policy information obtained, for example, from policy file **244**. Specifically, policy object **242** provides a mapping of code identifiers to permissions, and is constructed based on the instructions within policy file **244**. Within the policy object **242**, the code identifiers and their associated authorized permissions may be represented by data structures or objects.

Protection domain objects **250** are created on demand when new classes are received by code executor **210**. When a new class is received, domain mapper **248** determines whether a protection domain is already associated with the code identifier. The domain mapper maintains data indicating which protection domains have been created and the code identifiers associated with the protection domains. If a protection domain is already associated with the code identifier, the domain mapper adds a mapping of the new class and protection domain to a mapping of classes and protection domains maintained by the domain mapper **248**.

If a protection domain object is not associated with the code identifier of the new class, a new protection domain object is created and populated with permissions. The protection domain is populated with those permission that are mapped to the code identifier of the new class based on the mapping of code identifiers to permissions in the policy object. Finally, the domain mapper adds a mapping of the new class and protection domain to the mapping of classes and protection domains as previously described.

In other embodiments of the invention, instead of storing the mapping of classes to protection domains in a domain mapper object, the mapping is stored as static fields in the protection domain class. The protection domain class is the class to which protection domain objects belong. There is only one instance of a static field for a class no matter how many objects belong to the class. The data indicating which protection domains have been created and the code sources associated with the protection domains is stored in static fields of the protection domain class. Alternatively, a mapping between a class and protections domains associated with the class is stored as static fields in the class.

Static methods are used to access and update the static data mentioned above. Static methods are invoked on behalf of the entire class, and may be invoked without referencing a specific object.

EXEMPLARY CALL STACK

The permission objects, protection domain objects and policy objects described above are used to determine the access rights of a thread. According to an aspect of the invention, such access rights vary over time based on what code the thread is currently executing, and which code invoked the code that is currently executing. The sequence of calls that resulted in execution of the currently executing code of a thread is reflected in the call stack of the thread.

FIG. **3** illustrates an example of a call stack of a thread as it exists at a particular point in time. Reference to the exemplary call stack shall be made to explain the operation of a security mechanism that enforces access rights in a way that allows the rights of the thread to vary over time.

Referring to FIG. **3**, it is a block diagram that includes a call stack **308** associated with a thread **306** in which the

method **340-1** of an object **240-1** calls the method **340-2** of another object **240-2** that calls the method **340-3** of yet another object **240-3** that calls a check permission method **382** of an access controller object **380**.

Thread **306** is a thread executing on computer system **100**. Call stack **308** is a stack data structure representing a calling hierarchy of the methods invoked by thread **306** at any given instance. At the instance illustrated in FIG. **3**, call stack **308** contains a frame **310** for each invocation of a method by a thread and not exited by that thread.

Each frame **310** corresponds to the method that has been called but not exited by thread **306**. The relative positions of the frames on the call stack **308** reflect the invocation order of the methods that correspond to the frames. When a method is exited, the frame **310** that corresponds to the method is removed from the top of the call stack **308**. When a method is invoked, a frame corresponding to the method is added to the top of the call stack **308**.

Each frame contains information about the method and object that correspond to the frame. From this information the class of method can be determined by invoking a "get class" method provided for every object by the code executor **210**. From the mapping in domain mapper object **248**, the protection domain associated with the class, object, and method for given frame **310** can be determined.

For example, assume thread **306** invokes method **340-1**. While executing method **340-1** thread **306** invokes method **340-2**, while executing method **340-2** thread **306** invokes method **340-3**, and while executing method **340-3** thread **306** invokes method **382**. At this point, call stack **308** represents the calling hierarchy of methods as shown in FIG. **3**. Frame **310-4** corresponds to method **382**, frame **310-3** to method **340-3**, method **340-2** to frame **310-2**, and method **340-1** to frame **310-1**. When thread **306** exits method **382**, frame **310-4** is removed from the stack.

METHOD/PERMISSION RELATIONSHIPS

Each method on the call stack is associated with a set of permissions. The set of permissions for a given method is determined by the protection domain associated with the source from which the code for the given method was received. The relationship between methods, protection domains and permissions shall now be described with continued reference to FIG. **3**.

For the purposes of illustration, it shall be assumed that Object **240-1** corresponds to part of a user interface. The code associated with the class to which object **240-1** belongs is received from a remote source. File access for remote code by default is limited to a default directory in accordance with the security policy of the administrators of computer system **100**.

Protection domain object **250-1** is mapped to the class of object **240-1**. Protection domain object **250-1** is associated with two permissions, which are a permission to "write" to "e:/tmp" and to "read" from "e:/tmp." Therefore, method **340-1** is authorized to write and read from "e:/tmp".

Method **340-1** invokes method **340-2** of object **240-2**. Object **240-2** is a password manager that is associated with methods for managing passwords for computer system **100**. Passwords are contained in two files, "c:/sys/pwd" and "d:/sys/pwd." Method **340-2** is a method that adds a password to one of the files.

Protection domain object **250-2** is mapped to the class of object **240-2**. Accordingly, protection domain object **250-2** contains two permissions, a permission to "write" to "c:/sys/pwd" and a permission to write to "d:/sys/pwd."

Method **340-2** invokes method **340-3** of object **240-3**. Object **240-3** is a resource manager that contains methods for managing a system directory disk "d:/sys". Method **340-3** is a method to update a record in a file in the directory "d:/sys".

Protection domain object **250-3** is mapped to the class of object **240-3**. Accordingly, protection domain object **250-3** is associated with two permissions: a permission to "write" to "d:/sys/*" and a permission to read to "read" from "d:/sys/*".

While protection domain objects are used to organize and determine the access rights of a particular method, some mechanism must be provided to determine the access rights of a thread whose call stack contains multiple methods whose code arrived from multiple sources. According to one embodiment of the invention, this determination is performed by an access controller, as shall be described in greater detail hereafter.

EXEMPLARY ACCESS CONTROLLER

According to an embodiment of the invention, an access controller is used to determine whether a particular action may be performed by a principal. Specifically, before a resource management object accesses a resource, the resource management object (e.g. object **340-3**) invokes a check permission method of an access controller object **380**. In the illustrated example, the resource manager method **340-3** invokes a check permission method **382** of access controller object **380** to determine whether access to the password file is authorized. To make this determination, the check permission method **282** of the access controller **380** performs the steps that shall be described hereafter with reference to FIG. **4**.

DETERMINING WHETHER AN ACTION IS AUTHORIZED

According to an embodiment of the invention, an action is authorized if the permission required to perform the action is included in each protection domain associated with the thread when a request to determine an authorization is made. A permission is said to be included in a protection domain if that permission is encompassed by one or more permissions associated with the protection domain. For example, if an action requires permission to write to file in the "e:/tmp" directory, then that required permission is included in protection domain object **250-1** because protection domain object **250-1** is explicitly associated with that permission.

Assume that thread **306** is executing method **310-3** when thread **306** makes a request for a determination of whether an action is authorized by invoking the check permission method **382**. Assume further that thread **306** has invoked method **340-1**, method **340-2**, and method **310-3** has not exited them when thread **306** invoked method **382**. The protection domains associated with thread **306** when the request for a determination of authorization is made are represented by protection domain object **250-1**, protection domain object **250-2**, and protection domain object **250-3**.

Note that given the calling hierarchy present in the current example, the required permission to perform an action of writing to file "d:/sys/pwd" is not authorized for thread **306** because the required permission is not encompassed by the only permission included in protection domain object **250-1** (i.e. write to "e:/tmp").

PRIVILEDGED METHODS

Sometimes the need arises to authorize an action that a method performs irrespective of the protection domains

associated with the methods that precede the method in the calling hierarchy of a thread. Updating a password is an example of when such a need arises.

Specifically, because the security of a password file is critical, the permissions required to update the password file are limited to very few specialized protection domains. Typically, such protection domains are associated with methods of objects from code that is "trusted" and that provides its own security mechanisms. For example, a method for updating a password may require the old password of a user when updating that user with a new password.

Because permissions to update passwords are limited to code from limited sources, code from all other sources will not be allowed to update the passwords. This is true even in situations such as that shown in FIG. **3**, where the code from a remote source (method **340-1**) attempts to change the password by invoking the trusted code (method **340-3**) which has permission to update the password. Access is denied in these situations because at least one method in the calling hierarchy (method **340-1**) does not have the necessary permission.

According to one embodiment of the invention, a privilege mechanism is provided to allow methods that do not themselves have the permission to perform actions to nevertheless cause the actions to be performed by calling special "privileged" methods that do have the permissions. This result is achieved by limiting the protection domains that are considered to be "associated with a thread" to only those protection domains that are associated with a "privileged" method and those methods that are subsequent to the privileged method in the calling hierarchy.

A method may cause itself to be privileged (i.e. enable the privilege mechanism) by invoking a method of a privilege object called, for example, beginPrivilege. A method may cause itself to become not privileged (i.e. disable the privilege mechanism) by invoking another method of the privilege object called, for example, end privilege. The following code example illustrates one technique for invoking methods which enable or disable the privilege mechanism. Although the code example may resemble the JAVA programming language by Sun Microsystems Inc., the example is merely for illustrative purposes and is not meant to be representative of an actual code implementation.

```
Privileged p = new Privileged();
p.beginPrivilege();
try {
    [sensitive code]
} finally {
    p.endPrivilege();
}
```

The first line of the code example creates a privilege object. The second invokes a beginPrivilege method of the privilege object that enables the privilege mechanism. The "try finally" statement ensures that the block of code following the "finally" is executed regardless of what happens during execution of the block between the "try" and "finally". Thus the privilege disabling method of the privilege object ("p.endPrivilege( )") is always invoked.

The above code can be used, for example, to bound the portion of method **340-3** that actually accesses the password file. The portion that accesses the password file would be contained in the block designated as "[sensitive code]". The technique illustrated by the above code example explicitly places the responsibility of enabling and disabling the privilege mechanism upon the programmer.

Often, while executing a privileged method, a thread may invoke subsequent methods associated with other protection domains that do not include permissions included in the privileged protection domain. When a thread is executing a subsequent method, an action requested by the thread is only authorized if the required permission is encompassed in the protection domains associated with the subsequent method and any methods in the calling hierarchy between the subsequent method and privileged method, inclusively. The advantage of the limiting the privilege mechanism in this manner is to prevent methods of untrusted code from effectively "borrowing" the permissions associated with privileged methods of trusted code when the methods of the untrusted code are invoked by the trusted methods.

In an alternate embodiment of the invention, a method causes itself to be privileged or not privileged by invoking static methods of the access controller class. The access controller class is the class to which access controller objects belong. As demonstrated in the following code example, using static methods that are associated with the access controller class avoids the need of having to create a privilege object in order to enable the privilege mechanism.

The following code example illustrates one technique for invoking methods which enable or disable the privilege mechanism. Assume for purposes of illustration that the access controller class name is AccessControl. Although the code example may resemble the Java programming language by Sun Microsystems Inc., the example is merely for illustrative purposes and is not meant to be representative of an actual code implementation.

```
AccessControl.beginPrivilege();
try {
    [sensitive code]
} finally {
    AccessControl.endPrivilege();
}
```

ENABLING INVOCATIONS

A thread may invoke the same method at different levels in a calling hierarchy. For example, a method X may call a method Y which may call method X. Consequently, a method such as method **340-2** that is invoked as a privileged method could be invoked a second time without enabling the privilege mechanism in the second invocation. To properly determine the protection domains associated with a thread while the privilege mechanism is enabled, a mechanism is provided to track which invocation of the privileged method enabled the privilege mechanism. The invocation in which a thread enables the privilege mechanism is referred to as a enabling invocation.

One technique to track which invocations of a particular method are enabling invocations is to set a flag (e.g. privilege flag **312**) in the frame **310** corresponding to each enabling invocation. This may be accomplished by setting the privilege flag **312** in the frame corresponding to each enabling invocation. The flag may be set, for example, when the privilege enabling method of each privilege enabling object is invoked during execution of a method.

According to one embodiment of the invention, each frame has a privilege flag value. When any frame is added to the call stack **380**, the initial value of the privilege flag

indicates that the corresponding method is not privileged. The privilege flag of any frame is only set to a value indicating the corresponding method is privileged when the corresponding method enables the privilege. After a method that enables the privilege mechanism is exited, the value of the privilege flag **312** will not carry over to the next invocation of the method. The value will not carry over because when the new frame corresponding to the method is added to the call stack **308**, the initial value of the privilege flag is set to indicate that the corresponding method is not privileged. Maintaining the value of the privilege status flag in this manner disables the privilege mechanism when a privileged method is exited regardless of whether the privilege mechanism is explicitly disabled by the programmer.

While one method of tracking which invocations are enabling invocation is described above, various alternative methods of tracking enabling invocations are possible. Therefore, it is understood that the present invention is not limited to any specific method for tracking enabling invocations.

The method shown in FIG. **4** shall now be described with reference to the thread **306** and stack **308** illustrated in FIG. **3**. Assume for example, that thread **306** is executing a user interface method **340-1** to update a password. To update the password, thread **306** then invokes method **340-2** (the method to update a password), then method **340-3** (the method to update a file). Assume further that method **340-2** is privileged.

In step **410**, the request for a determination of whether an action is authorized is received. After detecting that a request for an action to update the password file has been detected by invoking method **340-3**, the permission required to perform the action is determined.

In the present example, the action is updating the password file and the required permission to perform the action is "write" to file "d:/sys/pwd". A request is made to determine whether the action is authorized by invoking the check permission method **382** of the access controller **380**, passing in as a parameter the permission required to perform the action. Note the current state of call stack **308** is shown in FIG. **3**.

Steps **430**, **440**, **450** and **460** define a loop in which permissions associated with the methods in the call stack are checked. The loop continues until a privileged method is encountered, or all of the methods in the call stack of have been checked. For the purposes of explanation, the method whose privileges are currently being checked is referred to as the "selected method".

In step **430**, a determination is made as to whether one of the permissions associated with the selected method encompasses the permission required. The permissions associated with a method are the permissions associated with the protection domain that is associated with the method. If the determination made in step **430** is that a permission associated with the selected method encompasses the permission required, control passes to step **440**.

During the first iteration of the loop, the frame that immediately precedes the frame associated with the check permission method of the access controller is inspected. In this example, the frame associated with the check permission method **382** is frame **310-4**. The frame that immediately precedes frame **310-4** is frame **310-3**. Consequently, during the first iteration of the loop, frame **310-3** will be inspected. Frame **310-3** is associated with method **340-3**, which is associated with protection domain **250-3**. Because a permission associated with protection domain object **250-3** (permission to "write" to "d:/sys/*") encompasses the permission required (i.e. "write" to "d:/sys/pwd"), control passes to step **440**.

In step **440**, a determination is made of whether the invocation of a selected method represents the enabling invocation. This determination is based on the privilege flag **312** of frame **310** corresponding to the invocation of the selected method. If the determination is that the invocation of the selected method does not represent the enabling invocation, control passes to step **450**. In this example, the privilege status of frame **310-3** is not set to indicate that the frame represents the enabling invocation, thus control passes to step **450**.

In steps **450**, the next method is selected. The next method is the method corresponding to frame below the current frame based on the calling hierarchy represented by call stack **308**. In this example, the frame below the current frame **310-3** is frame **310-2**. The method corresponding to frame **310-2** is method **340-2**.

In step **460**, a determination is made of whether a method was selected in step **450**. If a method was selected, control reverts to step **430**.

In the current example, control passes to step **430** because method **340-2** was selected. In step **430**, the determination made is that the protection domain associated with method **340-2** (protection domain object **250-2**) includes a permission encompassing the permission required ("write" to "d:/sys/pwd") because a permission associated with protection domain object **250-2** ("write" to "d:/sys/pwd") explicitly encompasses the permission required. Control passes to step **440**.

In step **440**, the determination made is that the invocation of a selected method represents the enabling invocation because the privilege flag **312** indicates that the invocation corresponding to frame **310-2** is an enabling invocation. A message is transmitted indicating the permission request is valid. Then, performance of the steps ends.

Note that by exiting the performance of the steps at step **440** when the selected method represents the enabling invocation, the authorization of the requested action is based on the privileged protection domain and any protection domains associated with methods invoked after the invocation of the enabling invocation.

Assume in the current example that the privilege mechanism was never invoked. Thus in step **440**, the determination made is that invocation of a selected method does not represent the enabling invocation because the privilege flag **312** indicates that the invocation corresponding to frame **310-2** is not an enabling invocation.

In steps **450**, the next method selected is method **340-1** because the frame below the current frame **310-2** is frame **310-1** and the method corresponding to frame **310-1** is method **340-1**. In step **460**, the determination made is that a next method was selected in step **450**, thus control reverts to step **430**.

In step **430**, the determination made is that the protection domain associated with method **340-1** (protection domain object **250-1**) does not include the permission required ("write" to "d:/sys/pwd") because no permission associated with protection domain object **250-1** (i.e. "e:/tmp" the only permission associated with the protection domain) encompasses the permission required. Control then passes to step **490**.

In step **490**, a message indicating that the requested action is not authorized is transmitted. In embodiment of the invention, the message is transmitted by throwing an exception error.

Note that when at least one protection domain associated with a thread does not include a permission encompassing the permission required, the action requested is not authorized. An action is authorized only when all the protection domains associated with a thread when making the request for a determination of whether the action is authorized include the permission required.

In one embodiment of the invention, when a thread ("parent thread") causes the spawning of another thread ("child thread"), the protection domains associated with the parent thread are "inherited" by the child thread. The protection domains may be inherited by, for example, retaining the call stack of a parent thread when the child thread is created. When the steps shown in FIG. **4** are executed to determine whether an action is authorized, the call stack that is traversed is treated as if it included the call stack of the parent thread. In another embodiment of the invention, a child thread does not inherit the protection domain of the parent thread, and, accordingly, the call stack that is traversed is treated as if it did not include the parent's call stack.

One advantage of basing the authorization of a thread to perform an action on the protection domains associated with the thread is that the permissions can be based on the source of code the thread is executing. As mentioned earlier, objects are created from class definitions in code received by code executor **210**. The source of code a thread is executing is the source of code of the method. The source of code of a method is the source of the class definition used to define the class to which the method's object belongs. Because the protection domains are associated with the source of code of a method, as described previously, the permissions authorized for a thread can be based on the source of the code of each method invoked by a thread. Thus, it can be organized so that code from a particular source is associated with the permissions appropriate for security purposes.

An advantage of the privilege mechanism described above is that performance of sensitive operations in which security is critical can be limited to methods from trusted sources. Furthermore, these operations can be performed on behalf of methods based on code from less secure sources. Methods performing sensitive operations typically rely on their own security mechanisms (e.g. password authentication methods). When a thread invokes the privilege mechanism, the scope of the permissions of privileged domain, which typically entail a high security risk, are limited to the enabling invocation. This prevents a method invoked within the privileged method, such as a method based on untrusted code, from acquiring the capability to perform operations posing a high security risk.

In the foregoing specification, the invention has been described with reference to specific embodiments thereof. It will, however, be evident that various modifications and changes may be made thereto without departing from the broader spirit and scope of the invention. The specification and drawings are, accordingly, to be regarded in an illustrative rather than a restrictive sense.

What is claimed is:

1. A method for providing security, the method comprising the steps of:

detecting when a request for an action is made by a principal; and

in response to detecting the request, determining whether said action is authorized based on permissions associated with a plurality of routines in a calling hierarchy associated with said principal, wherein said permissions are associated with said plurality of routines based on a first association between protection domains and permissions.

2. The method of claim **1**, wherein:

the step of detecting when a request for an action is made includes detecting when a request for an action is made by a thread; and

the step of determining whether said action is authorized includes determining whether said action is authorized based on an association between permissions and a plurality of routines in a calling hierarchy associated with said thread.

3. The method of claim **1**, wherein:

the calling hierarchy includes a first routine; and

the step of determining whether said action is authorized further includes determining whether a permission required to perform said action is encompassed by at least one permission associated with said first routine.

4. The method of claim **1**, wherein the step of determining whether said action is authorized further includes determining whether a permission required to perform said action is encompassed by at least one permission associated with each routine in said calling hierarchy.

5. A method for providing security, the method comprising the steps of:

detecting when a request for an action is made by a principal,

determining whether said action is authorized based on an association between permissions and a plurality of routines in a calling hierarchy associated with said principal;

wherein each routine of said plurality of routines is associated with a class; and

wherein said association between permissions and said plurality of routines is based on a second association between classes and protection domains.

6. A method for providing security, the method comprising the steps of:

detecting when a request for an action is made by a principal; and

in response to detecting the request, determining whether said action is authorized based on permissions associated with a plurality of routines in a calling hierarchy associated with said principal, wherein a first routine in said calling hierarchy is privileged; and

wherein the step of determining whether said action is authorized further includes determining whether a permission required to perform said action is encompassed by at least one permission associated with each routine in said calling hierarchy between and including said first routine and a second routine in said calling hierarchy, wherein said second routine is invoked after said first routine, wherein said second routine is a routine for performing said requested action.

7. The method of claim **6**, wherein the step of determining whether said permission required to perform said action is encompassed by at least one permission associated with each routine in said calling hierarchy between and including said first routine and said second routine further includes the steps of:

determining whether said permission required is encompassed by at least one permission associated with said second routine; and

in response to determining said permission required is encompassed by at least one permission associated with said second routine, then performing the steps of:

A) selecting a next routine from said plurality of routines in said calling hierarchy,

B) if said permission required is not encompassed by at least one permission associated with said next routine, then transmitting a message indicating that said permission required is not authorized, and

C) repeating steps A and B until:

said permission required is not authorized by at least one permission associated with said next routine,

there are no more routines to select from said plurality of routines in said calling hierarchy, or

determining that said next routine is said first routine.

8. The method of claim **7**, wherein:

the method further includes the step of setting a flag associated with said first routine to indicate that said first routine is privileged; and

the step of determining that said next routine is said first routine includes determining that a flag associated with said next routine indicates said next routine is privileged.

9. The method of claim **8**, wherein the step of setting said flag associated with said first routine includes setting a flag in a frame in said calling hierarchy associated with said thread.

10. A computer-readable medium carrying one or more sequences of one or more instructions, the one or more sequences of the one or more instructions including instructions which, when executed by one or more processors, causes the one or more processors to perform the steps of:

detecting when a request for an action is made by a principal; and

in response to detecting the request, determining whether said action is authorized based on permissions associated with a plurality of routines in a calling hierarchy associated with said principal, wherein said permissions are associated with said plurality of routines based on a first association between protection domains and permissions.

11. The computer-readable medium of claim **10**, wherein:

the step of detecting when a request for an action is made includes detecting when a request for an action is made by a thread; and

the step of determining whether said action is authorized includes determining whether said action is authorized based on an association between permissions and a plurality of routines in a calling hierarchy associated with said thread.

12. The computer readable medium of claim **10**, wherein:

the calling hierarchy includes a first routine; and

the step of determining whether said action is authorized further includes determining whether a permission required to perform said action is encompassed by at least one permission associated with said first routine.

13. The computer readable medium of claim **10**, wherein the step of determining whether said action is authorized further includes determining whether a permission required to perform said action is encompassed by at least one permission associated with each routine in said calling hierarchy.

14. A computer-readable medium bearing instructions for providing security, the instructions including instructions for performing the steps of:

detecting when a request for an action is made by a principal;

determining whether said action is authorized based on an association between permissions and a plurality of routines in a calling hierarchy associated with said principal;

wherein each routine of said plurality of routines is associated with a class; and

wherein said association between permissions and said plurality of routines is based on a second association between classes and protection domains.

15. A computer-readable medium carrying one or more sequences of one or more instructions, the one or more sequences of the one or more instructions including instructions which, when executed by one or more processors, causes the one or more processors to perform the steps of:

detecting when a request for an action is made by a principal; and

in response to detecting the request, determining whether said action is authorized based on permissions associated with a plurality of routines in a calling hierarchy associated with said principal, wherein a first routine in said calling hierarchy is privileged; and

wherein the step of determining whether said action is authorized further includes determining whether a permission required to perform said action is encompassed by at least one permission associated with each routine in said calling hierarchy between and including said first routine and a second routine in said calling hierarchy, wherein said second routine is invoked after said first routine, wherein said second routine is a routine for performing said requested action.

16. The computer readable medium of claim **15**, wherein the step of determining whether said permission required to perform said action is encompassed by at least one permission associated with each routine in said calling hierarchy between and including said first routine and said second routine further includes the steps of:

determining whether said permission required is encompassed by at least one permission associated with said second routine; and

in response to determining said permission required is encompassed by at least one permission associated with said second routine, then performing the steps of:

A) selecting a next routine from said plurality of routines in said calling hierarchy,

B) if said permission required is not encompassed by at least one permission associated with said next routine, then transmitting a message indicating that said permission required is not authorized, and

C) repeating steps A and B until:

said permission required is not authorized by at least one permission associated with said next routine,

there are no more routines to select from said plurality of routines in said calling hierarchy, or

determining that said next routine is said first routine.

17. The computer readable medium of claim **16**, wherein:

the computer readable medium further comprises one or more instructions for performing the step of setting a flag associated with said first routine to indicate that said first routine is privileged; and

the step of determining that said next routine is said first routine includes determining that a flag associated with said next routine indicates said next routine is privileged.

18. The computer readable medium of claim **17**, wherein the step of setting said flag associated with said first routine includes setting a flag in a frame in said calling hierarchy associated with said thread.

19. A computer system comprising:

a processor;

a memory coupled to said processor;

said processor being configured to detect when a request for an action is made by a principal; and

said processor being configured to respond to detecting the request by determining whether said action is authorized based on permissions associated with a plurality of routines in a calling hierarchy associated with said principal, wherein said permissions are associated with said plurality of routines based on a first association between protection domains and permissions.

20. The computer system of claim **19**, wherein:

the calling hierarchy includes a first routine; and

said processor is configured to determine whether said action is authorized by determining whether a permission required to perform said action is encompassed by at least one permission associated with said first routine.

21. The computer system of claim **19**, wherein

said processor is configured to determine whether said action is authorized by determining whether a permission required to perform said action is encompassed by at least one permission associated with each routine in said calling hierarchy.

* * * * *

EXHIBIT C

US005966702A

United States Patent [19]

Fresko et al.

[11] **Patent Number: 5,966,702**

[45] **Date of Patent: Oct. 12, 1999**

[54] **METHOD AND APPARATUS FOR PRE-PROCESSING AND PACKAGING CLASS FILES**

[75] Inventors: **Nedim Fresko**; **Richard Tuck**, both of San Francisco, Calif.

[73] Assignee: **Sun Microsystems, Inc.**, Palo Alto, Calif.

[21] Appl. No.: **08/961,874**

[22] Filed: **Oct. 31, 1997**

[51] **Int. Cl.**[6] **G06F 17/30**

[52] **U.S. Cl.** **707/1**; 707/7; 707/103; 707/10

[58] **Field of Search** 707/7, 103, 100, 707/102, 200, 10

[56] **References Cited**

U.S. PATENT DOCUMENTS

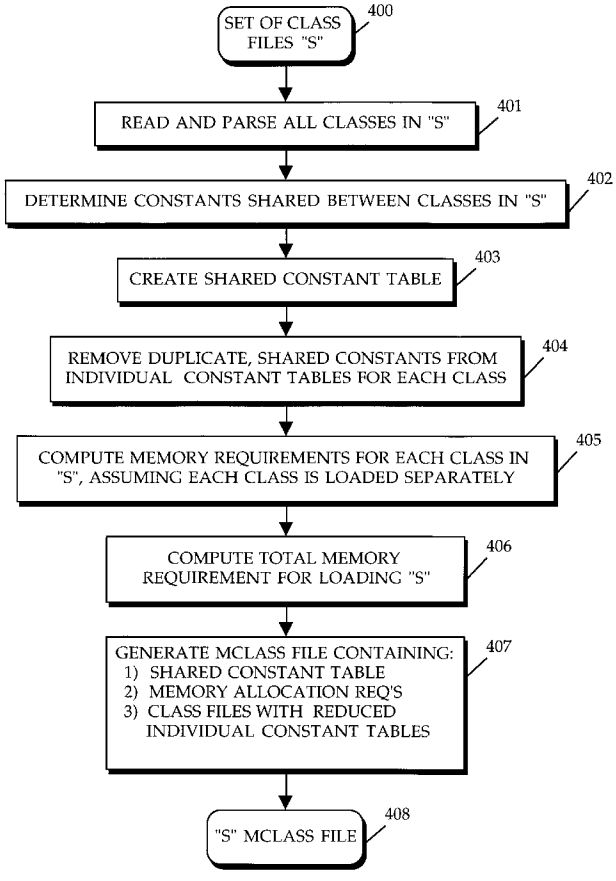
5,303,149	4/1994	Janigian	364/408
5,367,675	11/1994	Cheng et al.	395/600
5,488,725	1/1996	Turtle et al.	395/600
5,548,758	8/1996	Pirahesh et al.	395/600
5,717,915	2/1998	Stolfo et al.	395/605
5,732,265	3/1998	Dewitt et al.	395/616
5,813,009	9/1998	Johnson et al.	707/100
5,826,265	10/1998	Van Huben et al.	707/8
5,829,006	10/1998	Parvathaneny et al.	707/101
5,838,965	11/1998	Kavanagh et al.	395/614

Primary Examiner—Anton Fetting
Assistant Examiner—Michael J. Wallace, Jr.
Attorney, Agent, or Firm—Hecker & Harriman

[57] **ABSTRACT**

A method and apparatus for pre-processing and packaging class files. Embodiments remove duplicate information elements from a set of class files to reduce the size of individual class files and to prevent redundant resolution of the information elements. Memory allocation requirements are determined in advance for the set of classes as a whole to reduce the complexity of memory allocation when the set of classes are loaded. The class files are stored in a single package for efficient storage, transfer and processing as a unit. In an embodiment, a pre-processor examines each class file in a set of class files to locate duplicate information in the form of redundant constants contained in a constant pool. The duplicate constant is placed in a separate shared table, and all occurrences of the constant are removed from the respective constant pools of the individual class files. During pre-processing, memory allocation requirements are determined for each class file, and used to determine a total allocation requirement for the set of class files. The shared table, the memory allocation requirements and the reduced class files are packaged as a unit in a multi-class file.

23 Claims, 6 Drawing Sheets



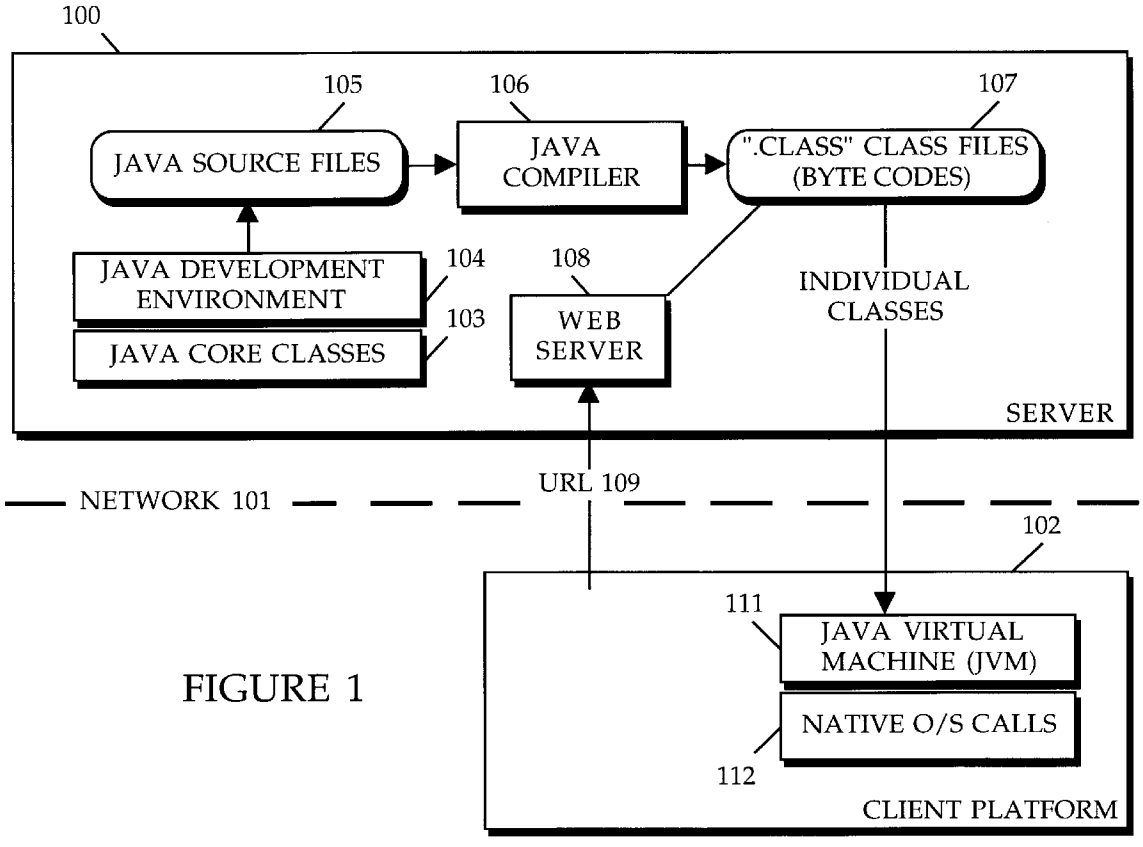

100
105
106
107
JAVA SOURCE FILES
JAVA COMPILER
".CLASS" CLASS FILES (BYTE CODES)
JAVA DEVELOPMENT ENVIRONMENT
104
103
JAVA CORE CLASSES
108
WEB SERVER
INDIVIDUAL CLASSES
SERVER
URL 109
NETWORK 101
102
111
JAVA VIRTUAL MACHINE (JVM)
NATIVE O/S CALLS
112
CLIENT PLATFORM

FIGURE 1

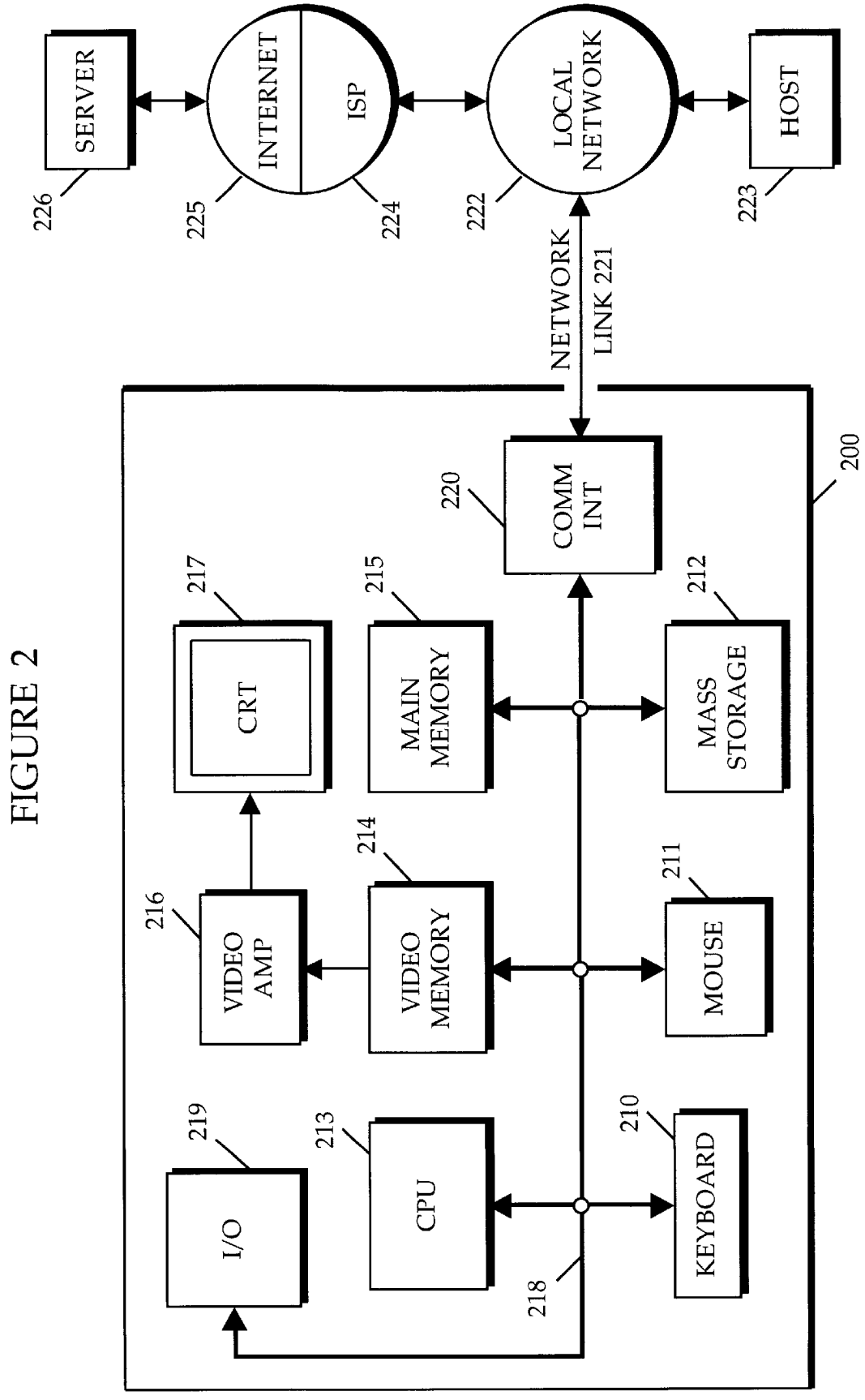

SERVER
INTERNET
ISP
LOCAL NETWORK
HOST
226
225
224
222
223
NETWORK LINK 221
200
COMM INT
220
CRT
217
MAIN MEMORY
215
MASS STORAGE
212
VIDEO AMP
216
VIDEO MEMORY
214
MOUSE
211
I/O
219
CPU
213
KEYBOARD
210
218

FIGURE 2

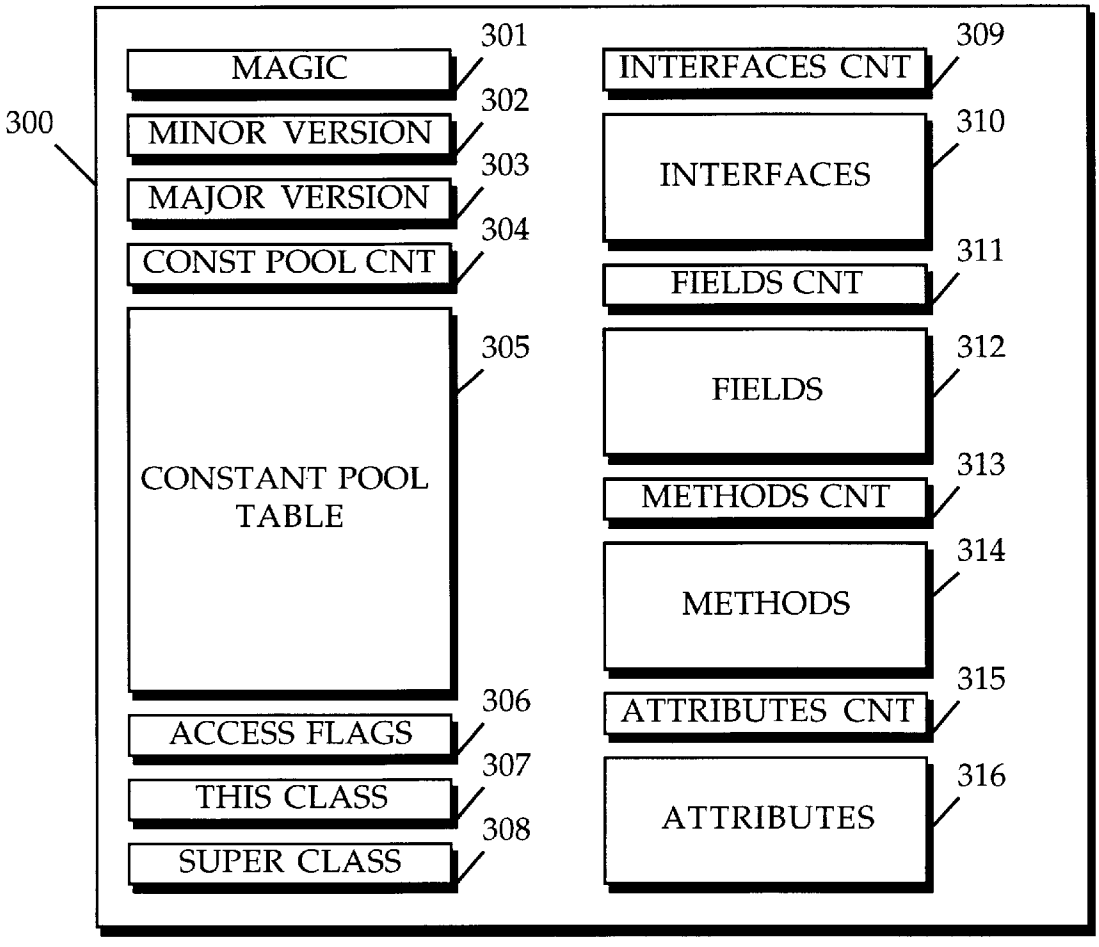
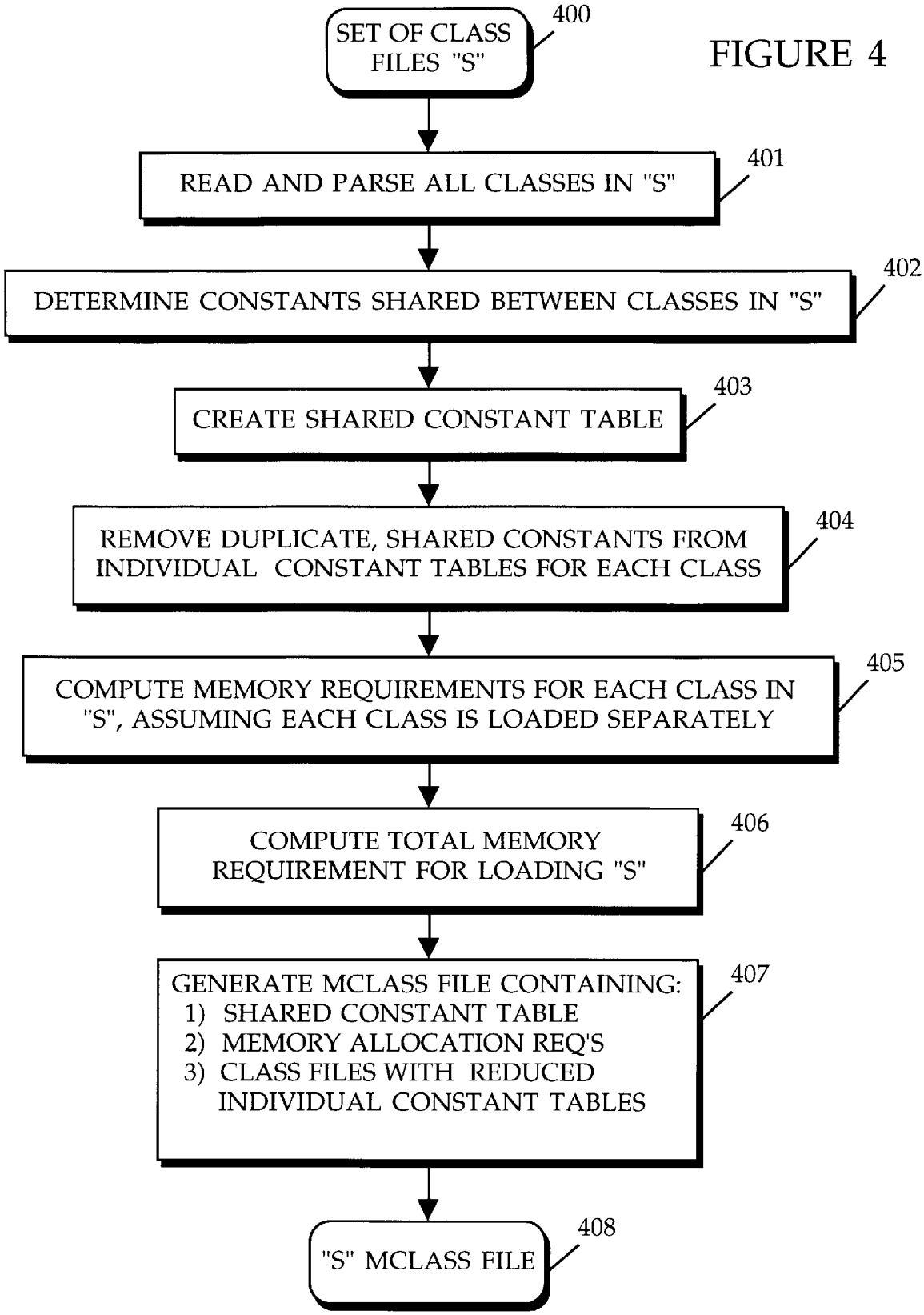

300
MAGIC
301
MINOR VERSION
302
MAJOR VERSION
303
CONST POOL CNT
304
CONSTANT POOL TABLE
305
ACCESS FLAGS
306
THIS CLASS
307
SUPER CLASS
308
INTERFACES CNT
309
INTERFACES
310
FIELDS CNT
311
FIELDS
312
METHODS CNT
313
METHODS
314
ATTRIBUTES CNT
315
ATTRIBUTES
316

FIGURE 3

FIGURE 4


SET OF CLASS FILES "S"
400
READ AND PARSE ALL CLASSES IN "S"
401
DETERMINE CONSTANTS SHARED BETWEEN CLASSES IN "S"
402
CREATE SHARED CONSTANT TABLE
403
REMOVE DUPLICATE, SHARED CONSTANTS FROM INDIVIDUAL CONSTANT TABLES FOR EACH CLASS
404
COMPUTE MEMORY REQUIREMENTS FOR EACH CLASS IN "S", ASSUMING EACH CLASS IS LOADED SEPARATELY
405
COMPUTE TOTAL MEMORY REQUIREMENT FOR LOADING "S"
406
GENERATE MCLASS FILE CONTAINING:
1) SHARED CONSTANT TABLE
2) MEMORY ALLOCATION REQ'S
3) CLASS FILES WITH REDUCED INDIVIDUAL CONSTANT TABLES
407
"S" MCLASS FILE
408

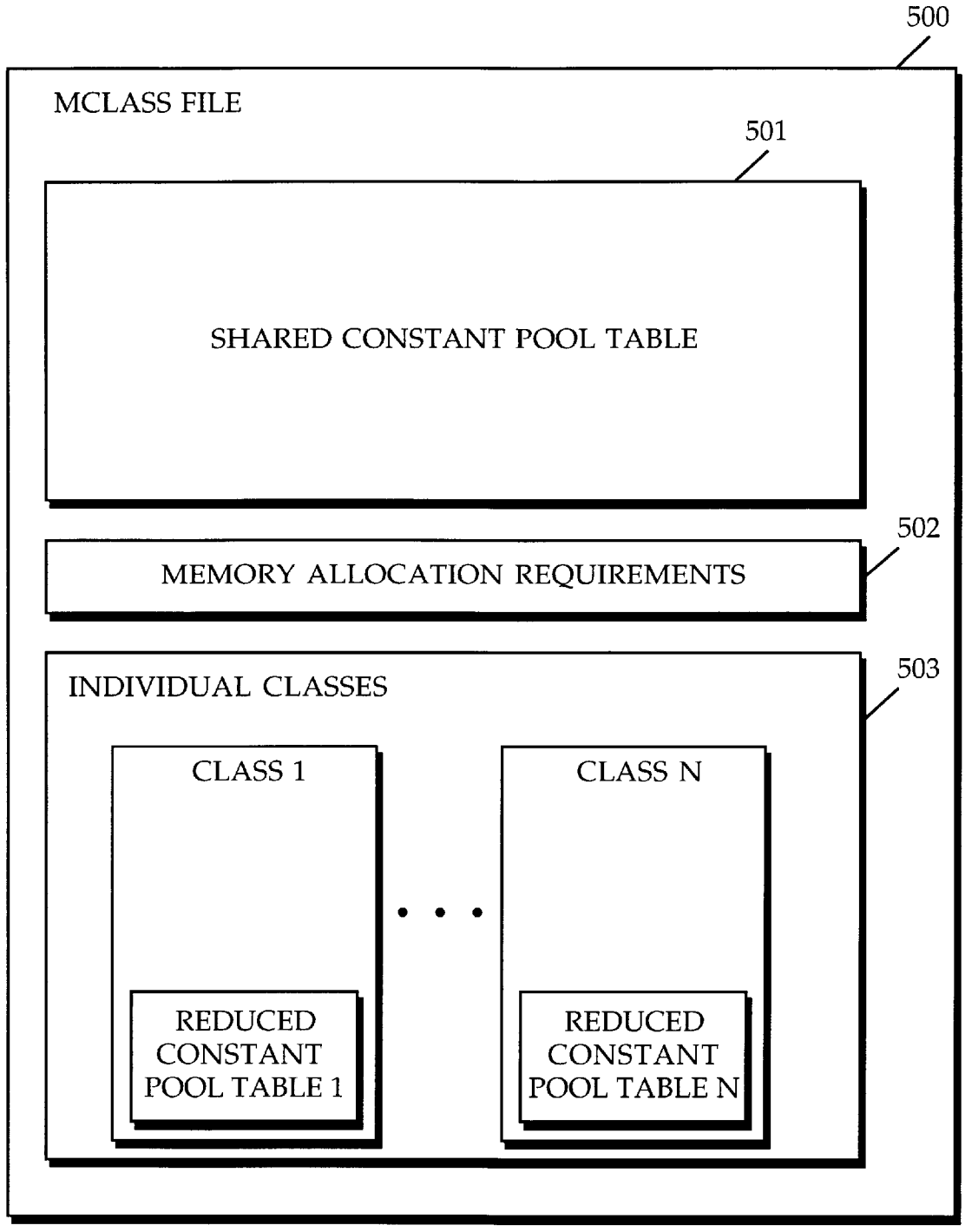

500
MCLASS FILE
501
SHARED CONSTANT POOL TABLE
502
MEMORY ALLOCATION REQUIREMENTS
503
INDIVIDUAL CLASSES
CLASS 1
CLASS N
REDUCED CONSTANT POOL TABLE 1
REDUCED CONSTANT POOL TABLE N

FIGURE 5

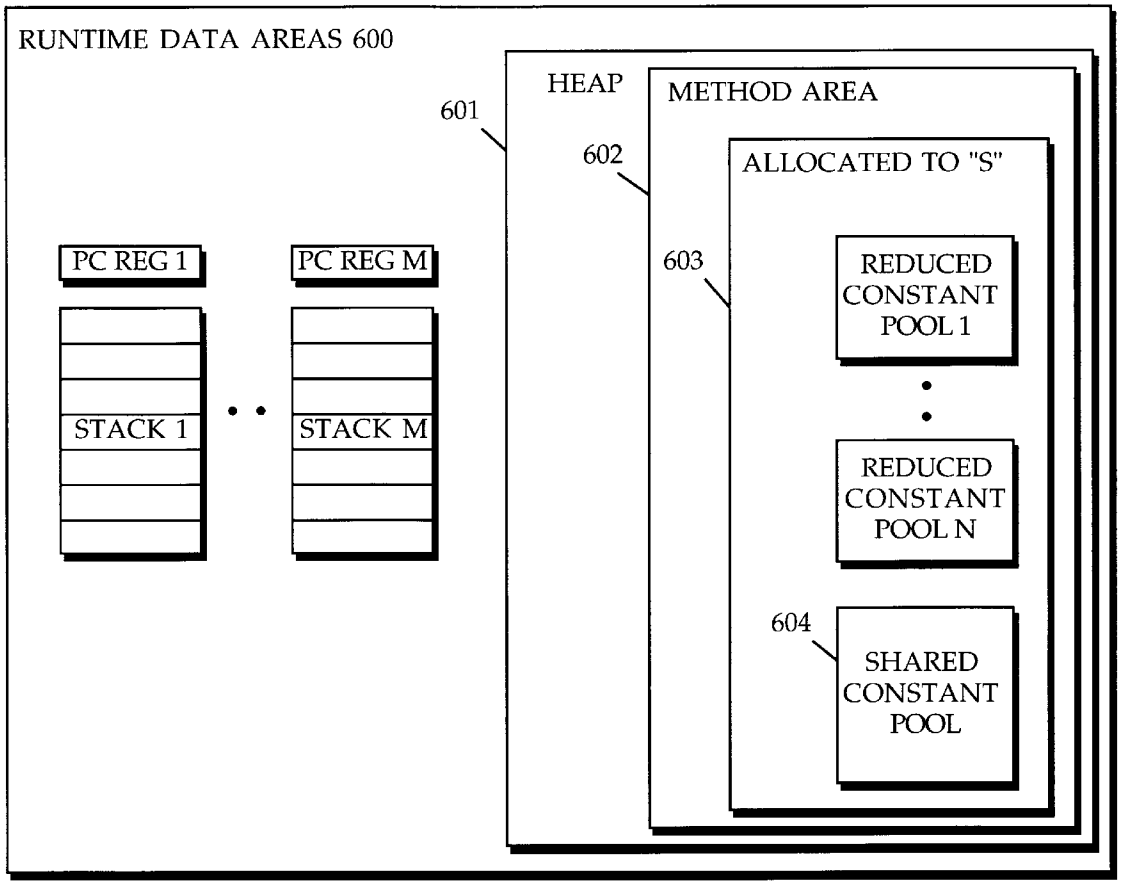

RUNTIME DATA AREAS 600
HEAP
601
METHOD AREA
602
ALLOCATED TO "S"
603
PC REG 1
PC REG M
STACK 1
STACK M
REDUCED CONSTANT POOL 1
REDUCED CONSTANT POOL N
604
SHARED CONSTANT POOL

FIGURE 6

METHOD AND APPARATUS FOR PRE-PROCESSING AND PACKAGING CLASS FILES

BACKGROUND OF THE INVENTION

1. Field of the Invention

This invention relates to the field of computer software, and, more specifically, to object-oriented computer applications.

Portions of the disclosure of this patent document contain material that is subject to copyright protection. The copyright owner has no objection to the facsimile reproduction by anyone of the patent document or the patent disclosure as it appears in the Patent and Trademark Office file or records, but otherwise reserves all copyright rights whatsoever.

2. Background Art

With advancements in network technology, the use of networks for facilitating the distribution of media information, such as text, graphics, and audio, has grown dramatically, particularly in the case of the Internet and the World Wide Web. One area of focus for current developmental efforts is in the field of web applications and network interactivity. In addition to passive media content, such as HTML definitions, computer users or "clients" coupled to the network are able to access or download application content, in the form of applets, for example, from "servers" on the network.

To accommodate the variety of hardware systems used by clients, applications or applets are distributed in a platform-independent format such as the Java® class file format. Object-oriented applications are formed from multiple class files that are accessed from servers and downloaded individually as needed. Class files contain bytecode instructions. A "virtual machine" process that executes on a specific hardware platform loads the individual class files and executes the bytecodes contained within.

A problem with the class file format and the class loading process is that class files often contain duplicated data. The storage, transfer and processing of the individual class files is thus inefficient due to the redundancy of the information. Also, an application may contain many class files, all of which are loaded and processed in separate transactions. This slows down the application and degrades memory allocator performance. Further, a client is required to maintain a physical connection to the server for the duration of the application in order to access class files on demand.

These problems can be understood from a review of general object-oriented programming and an example of a current network application environment.

Object-Oriented Programming

Object-oriented programming is a method of creating computer programs by combining certain fundamental building blocks, and creating relationships among and between the building blocks. The building blocks in object-oriented programming systems are called "objects." An object is a programming unit that groups together a data structure (one or more instance variables) and the operations (methods) that can use or affect that data. Thus, an object consists of data and one or more operations or procedures that can be performed on that data. The joining of data and operations into a unitary building block is called "encapsulation."

An object can be instructed to perform one of its methods when it receives a "message." A message is a command or

instruction sent to the object to execute a certain method. A message consists of a method selection (e.g., method name) and a plurality of arguments. A message tells the receiving object what operations to perform.

One advantage of object-oriented programming is the way in which methods are invoked. When a message is sent to an object, it is not necessary for the message to instruct the object how to perform a certain method. It is only necessary to request that the object execute the method. This greatly simplifies program development.

Object-oriented programming languages are predominantly based on a "class" scheme. The class-based object-oriented programming scheme is generally described in Lieberman, "Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems," OOPSLA 86 Proceedings, September 1986, pp. 214–223.

A class defines a type of object that typically includes both variables and methods for the class. An object class is used to create a particular instance of an object. An instance of an object class includes the variables and methods defined for the class. Multiple instances of the same class can be created from an object class. Each instance that is created from the object class is said to be of the same type or class.

To illustrate, an employee object class can include "name" and "salary" instance variables and a "set_salary" method. Instances of the employee object class can be created, or instantiated for each employee in an organization. Each object instance is said to be of type "employee." Each employee object instance includes "name" and "salary" instance variables and the "set_salary" method. The values associated with the "name" and "salary" variables in each employee object instance contain the name and salary of an employee in the organization. A message can be sent to an employee's employee object instance to invoke the "set_salary" method to modify the employee's salary (i.e., the value associated with the "salary" variable in the employee's employee object).

A hierarchy of classes can be defined such that an object class definition has one or more subclasses. A subclass inherits its parent's (and grandparent's etc.) definition. Each subclass in the hierarchy may add to or modify the behavior specified by its parent class. Some object-oriented programming languages support multiple inheritance where a subclass may inherit a class definition from more than one parent class. Other programming languages support only single inheritance, where a subclass is limited to inheriting the class definition of only one parent class. The Java programming language also provides a mechanism known as an "interface" which comprises a set of constant and abstract method declarations. An object class can implement the abstract methods defined in an interface. Both single and multiple inheritance are available to an interface. That is, an interface can inherit an interface definition from more than one parent interface.

An object is a generic term that is used in the object-oriented programming environment to refer to a module that contains related code and variables. A software application can be written using an object-oriented programming language whereby the program's functionality is implemented using objects.

A Java program is composed of a number of classes and interfaces. Unlike many programming languages, in which a program is compiled into machine-dependent, executable program code, Java classes are compiled into machine independent bytecode class files. Each class contains code and data in a platform-independent format called the class

file format. The computer system acting as the execution vehicle contains a program called a virtual machine, which is responsible for executing the code in Java classes. The virtual machine provides a level of abstraction between the machine independence of the bytecode classes and the machine-dependent instruction set of the underlying computer hardware. A "class loader" within the virtual machine is responsible for loading the bytecode class files as needed, and either an interpreter executes the bytecodes directly, or a "just-in-time" (JIT) compiler transforms the bytecodes into machine code, so that they can be executed by the processor. FIG. **1** is a block diagram illustrating a sample Java network environment comprising a client platform **102** coupled over a network **101** to a server **100** for the purpose of accessing Java class files for execution of a Java application or applet.

Sample Java Network Application Environment

In FIG. **1**, server **100** comprises Java development environment **104** for use in creating the Java class files for a given application. The Java development environment **104** provides a mechanism, such as an editor and an applet viewer, for generating class files and previewing applets. A set of Java core classes **103** comprise a library of Java classes that can be referenced by source files containing other/new Java classes. From Java development environment **104**, one or more Java source files **105** are generated. Java source files **105** contain the programmer readable class definitions, including data structures, method implementations and references to other classes. Java source files **105** are provided to Java compiler **106**, which compiles Java source files **105** into compiled ".class" files **107** that contain bytecodes executable by a Java virtual machine. Bytecode class files **107** are stored (e.g., in temporary or permanent storage) on server **100**, and are available for download over network **101**.

Client platform **102** contains a Java virtual machine (JVM) **111** which, through the use of available native operating system (O/S) calls **112**, is able to execute bytecode class files and execute native O/S calls when necessary during execution.

Java class files are often identified in applet tags within an HTML (hypertext markup language) document. A web server application **108** is executed on server **100** to respond to HTTP (hypertext transport protocol) requests containing URLs (universal resource locators) to HTML documents, also referred to as "web pages." When a browser application executing on client platform **102** requests an HTML document, such as by forwarding URL **109** to web server **108**, the browser automatically initiates the download of the class files **107** identified in the applet tag of the HTML document. Class files **107** are typically downloaded from the server and loaded into virtual machine **111** individually as needed.

It is typical for the classes of a Java program to be loaded as late during the program's execution as possible; they are loaded on demand from the network (stored on a server), or from a local file system, when first referenced during the Java program's execution. The virtual machine locates and loads each class file, parses the class file format, allocates memory for the class's various components, and links the class with other already loaded classes. This process makes the code in the class readily executable by the virtual machine.

The individualized class loading process, as it is typically executed, has disadvantages with respect to use of storage resources on storage devices, allocation of memory, and execution speed and continuity. Those disadvantages are magnified by the fact that a typical Java application can contain hundreds or thousands of small class files. Each class file is self-contained. This often leads to information redundancy between class files, for example, with two or more class files sharing common constants. As a result, multiple classes inefficiently utilize large amounts of storage space on permanent storage devices to separately store duplicate information. Similarly, loading each class file separately causes unnecessary duplication of information in application memory as well. Further, because common constants are resolved separately per class during the execution of Java code, the constant resolution process is unnecessarily repeated.

Because classes are loaded one by one, each small class requires a separate set of dynamic memory allocations. This creates memory fragmentation, which wastes memory, and degrades allocator performance. Also, separate loading "transactions" are required for each class. The virtual machine searches for a class file either on a network device, or on a local file system, and sets up a connection to load the class and parse it. This is a relatively slow process, and has to be repeated for each class. The execution of a Java program is prone to indeterminate pauses in response/execution caused by each class loading procedure, especially, when loading classes over a network. These pauses create a problem for systems in which interactive or real-time performance is important.

A further disadvantage of the individual class loading process is that the computer executing the Java program must remain physically connected to the source of Java classes during the duration of the program's execution. This is a problem especially for mobile or embedded computers without local disk storage or dedicated network access. If the physical connection is disrupted during execution of a Java application, class files will be inaccessible and the application will fail when a new class is needed. Also, it is often the case that physical connections to networks such as the Internet have a cost associated with the duration of such a connection. Therefore, in addition to the inconvenience associated with maintaining a connection throughout application execution, there is added cost to the user as a result of the physical connection.

A Java archive (JAR) format has been developed to group class files together in a single transportable package known as a JAR file. JAR files encapsulate Java classes in archived, compressed format. A JAR file can be identified in an HTML document within an applet tag. When a browser application reads the HTML document and finds the applet tag, the JAR file is downloaded to the client computer and decompressed. Thus, a group of class files may be downloaded from a server to a client in one download transaction. After downloading and decompressing, the archived class files are available on the client system for individual loading as needed in accordance with standard class loading procedures. The archived class files remain subject to storage inefficiencies due to duplicated data between files, as well as memory fragmentation due to the performance of separate memory allocations for each class file.

SUMMARY OF THE INVENTION

A method and apparatus for pre-processing and packaging class files is described. Embodiments of the invention remove duplicate information elements from a set of class files to reduce the size of individual class files and to prevent redundant resolution of the information elements. Memory

allocation requirements are determined in advance for the set of classes as a whole to reduce the complexity of memory allocation when the set of classes are loaded. The class files are stored in a single package for efficient storage, transfer and processing as a unit.

In an embodiment of the invention, a pre-processor examines each class file in a set of class files to locate duplicate information in the form of redundant constants contained in a constant pool. The duplicate constant is placed in a separate shared table, and all occurrences of the constant are removed from the respective constant pools of the individual class files. During pre-processing, memory allocation requirements are determined for each class file, and used to determine a total allocation requirement for the set of class files. The shared table, the memory allocation requirements and the reduced class files are packaged as a unit in a multi-class file.

When a virtual machine wishes to load the classes in the multi-class file, the location of the multi-class file is determined and the multi-class file is downloaded from a server, if needed. The memory allocation information in the multi-class file is used by the virtual machine to allocate memory from the virtual machine's heap for the set of classes. The individual classes, with respective reduced constant pools, are loaded, along with the shared table, into the virtual machine. Constant resolution is carried out on demand on the respective reduced constant pools and the shared table.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. **1** is an embodiment of a Java network application environment.

FIG. **2** is a block diagram of an embodiment of a computer system capable of providing a suitable execution environment for an embodiment of the invention.

FIG. **3** is a block diagram of an embodiment of a class file format.

FIG. **4** is a flow diagram of a class file pre-processing method in accordance with an embodiment of the invention.

FIG. **5** is a block diagram of an multi-class file format in accordance with an embodiment of the invention.

FIG. **6** is a block diagram of the runtime data areas of a virtual machine in accordance with an embodiment of the invention.

DETAILED DESCRIPTION OF THE INVENTION

The invention is a method and apparatus for pre-processing and packaging class files. In the following description, numerous specific details are set forth to provide a more thorough description of embodiments of the invention. It will be apparent, however, to one skilled in the art, that the invention may be practiced without these specific details. In other instances, well known features have not been described in detail so as not to obscure the invention.

Embodiment of Computer Execution Environment (Hardware)

An embodiment of the invention can be implemented as computer software in the form of computer readable program code executed on a general purpose computer such as computer **200** illustrated in FIG. **2**, or in the form of bytecode class files executable by a virtual machine running on such a computer. A keyboard **210** and mouse **211** are coupled to a bi-directional system bus **218**. The keyboard and mouse are for introducing user input to the computer system and communicating that user input to central processing unit (CPU) **213**. Other suitable input devices may be used in addition to, or in place of, the mouse **211** and keyboard **210**. I/O (input/output) unit **219** coupled to bi-directional system bus **218** represents such I/O elements as a printer, A/V (audio/video) I/O, etc.

Computer **200** includes a video memory **214**, main memory **215** and mass storage **212**, all coupled to bidirectional system bus **218** along with keyboard **210**, mouse **211** and CPU **213**. The mass storage **212** may include both fixed and removable media, such as magnetic, optical or magnetic optical storage systems or any other available mass storage technology. Bus **218** may contain, for example, thirty-two address lines for addressing video memory **214** or main memory **215**. The system bus **218** also includes, for example, a 32-bit data bus for transferring data between and among the components, such as CPU **213**, main memory **215**, video memory **214** and mass storage **212**. Alternatively, multiplex data/address lines may be used instead of separate data and address lines.

In one embodiment of the invention, the CPU **213** is a microprocessor manufactured by Motorola®, such as the 680X0 processor or a microprocessor manufactured by Intel®, such as the 80X86, or Pentium® processor, or a SPARC® microprocessor from Sun Microsystems®. However, any other suitable microprocessor or microcomputer may be utilized. Main memory **215** is comprised of dynamic random access memory (DRAM). Video memory **214** is a dual-ported video random access memory. One port of the video memory **214** is coupled to video amplifier **216**. The video amplifier **216** is used to drive the cathode ray tube (CRT) raster monitor **217**. Video amplifier **216** is well known in the art and may be implemented by any suitable apparatus. This circuitry converts pixel data stored in video memory **214** to a raster signal suitable for use by monitor **217**. Monitor **217** is a type of monitor suitable for displaying graphic images.

Computer **200** may also include a communication interface **220** coupled to bus **218**. Communication interface **220** provides a two-way data communication coupling via a network link **221** to a local network **222**. For example, if communication interface **220** is an integrated services digital network (ISDN) card or a modem, communication interface **220** provides a data communication connection to the corresponding type of telephone line, which comprises part of network link **221**. If communication interface **220** is a local area network (LAN) card, communication interface **220** provides a data communication connection via network link **221** to a compatible LAN. Wireless links are also possible. In any such implementation, communication interface **220** sends and receives electrical, electromagnetic or optical signals which carry digital data streams representing various types of information.

Network link **221** typically provides data communication through one or more networks to other data devices. For example, network link **221** may provide a connection through local network **222** to host computer **223** or to data equipment operated by an Internet Service Provider (ISP) **224**. ISP **224** in turn provides data communication services through the world wide packet data communication network now commonly referred to as the "Internet" **225**. Local network **222** and Internet **225** both use electrical, electromagnetic or optical signals which carry digital data streams. The signals through the various networks and the signals on network link **221** and through communication interface **220**, which carry the digital data to and from computer **200**, are exemplary forms of carrier waves transporting the information.

Computer **200** can send messages and receive data, including program code, through the network(s), network link **221**, and communication interface **220**. In the Internet example, server **226** might transmit a requested code for an application program through Internet **225**, ISP **224**, local network **222** and communication interface **220**. In accord with the invention, one such downloaded application is the apparatus for pre-processing and packaging class files described herein.

The received code may be executed by CPU **213** as it is received, and/or stored in mass storage **212**, or other non-volatile storage for later execution. In this manner, computer **200** may obtain application code in the form of a carrier wave.

The computer systems described above are for purposes of example only. An embodiment of the invention may be implemented in any type of computer system or programming or processing environment.

Class File Structure

Embodiments of the invention can be better understood with reference to aspects of the class file format. Description is provided below of the Java class file format. Also, enclosed as Section A of this specification are Chapter 4, "The class File Format," and Chapter 5, "Constant Pool Resolution," of *The Java Virtual Machine Specification*, by Tim Lindholm and Frank Yellin, published by Addison-Wesley in September 1996, ©Sun Microsystems, Inc.

The Java class file consists of a stream of 8-bit bytes, with 16-bit, 32-bit and 64-bit structures constructed from consecutive 8-bit bytes. A single class or interface file structure is contained in the class file. This class file structure appears as follows:

```
ClassFile    {
        u4 magic;
        u2 minor_version;
        u2 major_version;
        u2 constant_pool_count;
        cp_info constant_pool[constant_pool_count-1];
        u2 access_flags;
        u2 this_class;
        u2 super_class;
        u2 interfaces_count;
        u2 interfaces[interfaces_count];
        u2 fields_count;
        field_info fields[fields_count];
        u2 methods_count;
        method_info methods[methods_count];
        u2 attributes_count;
        attribute_info attributes[attributes_count];
}
```

where u2 and u4 refer to unsigned two-byte and four-byte quantities. This structure is graphically illustrated in FIG. **3**.

In FIG. **3**, class file **300** comprises four-byte magic value **301**, two-byte minor version number **302**, two-byte major version number **303** , two-byte constant pool count value **304**, constant pool table **305** corresponding to the constant pool array of variable length elements, two-byte access flags value **306**, two-byte "this class" identifier **307**, two-byte super class identifier **308**, two-byte interfaces count value **309**, interfaces table **310** corresponding to the interfaces array of two-byte elements, two-byte fields count value **311**, fields table **312** corresponding to the fields array of variable length elements, two-byte methods count value **313**, methods table **314** corresponding to the methods array of variable length elements, two-byte attributes count value **315**, and attributes table **316** corresponding to the attributes array of variable-length elements. Each of the above structures is briefly described below.

Magic value **301** contains a number identifying the class file format. For the Java class file format, the magic number has the value 0xCAFEBABE. The minor version number **302** and major version number **303** specify the minor and major version numbers of the compiler responsible for producing the class file.

The constant pool count value **304** identifies the number of entries in constant pool table **305**. Constant pool table **305** is a table of variable-length data structures representing various string constants, numerical constants, class names, field names, and other constants that are referred to within the ClassFile structure. Each entry in the constant pool table has the following general structure:

```
cp_info    {
        u1 tag;
        u1 info[ ];
}
```

where the one-byte "tag" specifies a particular constant type. The format of the info[ ] array differs based on the constant type. The info[ ] array may be a numerical value such as for integer and float constants, a string value for a string constant, or an index to another entry of a different constant type in the constant pool table. Further details on the constant pool table structure and constant types are available in Chapter 4 of Section A.

Access flags value **306** is a mask of modifiers used with class and interface declarations. The "this class" value **307** is an index into constant pool table **305** to a constant type structure representing the class or interface defined by this class file. The super class value **308** is either zero, indicating the class is a subclass of java.lang.Object, or an index into the constant pool table to a constant type structure representing the superclass of the class defined by this class file.

Interfaces count value **309** identifies the number of direct superinterfaces of this class or interface, and accordingly, the number of elements in interfaces table **310**. Interfaces table **310** contains two-byte indices into constant pool table **305**. Each corresponding entry in constant pool table **305** is a constant type structure representing an interface which is a direct superinterface of the class or interface defined by this class file.

The fields count value **311** provides the number of structures in fields table **312**. Each entry in fields table **312** is a variable-length structure providing a description of a field in the class type. Fields table **312** includes only those fields that are declared by the class or interface defined by this class file.

The methods count value **313** indicates the number of structures in methods table **314**. Each element of methods table **314** is a variable-length structure giving a description of, and virtual machine code for, a method in the class or interface.

The attributes count value **315** indicates the number of structures in attributes table **316**. Each element in attributes table **316** is a variable-length attribute structure. Attribute structures are discussed in section 4.7 of Section A.

Embodiments of the invention examine the constant pool table for each class in a set of classes to determine where duplicate information exists. For example, where two or more classes use the same string constant, the string constant may be removed from each class file structure and placed in a shared constant pool table. In the simple case, if N classes have the same constant entry, N units of memory space are

taken up in storage resources. By removing all constant entries and providing one shared entry, N−1 units of memory space are freed. The memory savings increase with N. Also, by implementing a shared constant table, entries in the constant table need be fully resolved at most once. After the initial resolution, future code references to the constant may directly use the constant.

Pre-processing and Packaging Classes

An embodiment of the invention uses a class pre-processor to package classes in a format called an "mclass" or multi-class file. A method for pre-processing and packaging a set of class files is illustrated in the flow diagram of FIG. **4**.

The method begins in step **400** with a set of arbitrary class files "S" (typically part of one application). In step **401**, the pre-processor reads and parses each class in "S." In step **402**, the pre-processor examines the constant pool tables of each class to determine the set of class file constants (such as strings and numerics, as well as others specific to the class file format) that can be shared between classes in "S." A shared constant pool table is created in step **403**, with all duplicate constants determined from step **402**. In step **404**, the pre-processor removes the duplicate, shared constants from the individual constant pool tables of each class.

In step **405**, the pre-processor computes the in-core memory requirements of each class in "S," as would normally be determined by the class loader for the given virtual machine. This is the amount of memory the virtual machine would allocate for each class, if it were to load each class separately. After considering all classes in "S" and the additional memory requirement for the shared constant pool table, the total memory requirement for loading "S" is computed in step **406**.

In step **407**, the pre-processor produces a multi-class (mclass) file that contains the shared constant pool table created in step **403**, information about memory allocation requirements determined in steps **405** and **406**, and all classes in "S," with their respective reduced constant pool tables. The mclass file for the class set "S" is output in step **408**. In some embodiments, to further reduce the size of the multi-class file, the multi-class file may be compressed.

An example of one embodiment of a multi-class file structure may be represented as follows:

```
MclassFile    {
        u2 shared_pool_count;
        cp_info shared_pool[shared_pool_count-1];
        u2 mem_alloc_req;
        u2 classfile_count;
        ClassFile classfiles[classfile_count];
}
```

In one embodiment of the invention, a new constant type is defined with a corresponding constant type tag. The new constant type provides as its info[ ] element an index into the shared constant table. During pre-processing, duplicated constant elements are placed in the shared constant pool as a shared element, and an element of the new constant type replaces the duplicated element in the reduced pool to direct constant resolution to the shared element in the shared constant pool. Reduction occurs because the replacement element is just a pointer to the actual constant placed in the shared constant pool.

FIG. **5** is a simplified block diagram of an embodiment of the multi-class file format. Mclass file **500** comprises shared constant pool table **501**, memory allocation requirements **502** and the set of individual classes **503**. The set of individual classes **503** comprises the class file structures for classes 1–N (N being the number of classes in the set), along with the corresponding reduced constant pool tables 1–N. The size of the shared constant pool table **501** is dependent on the number of duplicate constants found in the set of classes. The memory allocation requirements **502** may be represented as a single value indicating the total memory needed to load all class structures (classes 1–N) in individual classes **503**, as well as the shared constant pool table **501**. The shared pool count and classfile count (not shown in FIG. **5**) identify the number of elements in the shared constant pool table **501** and the classfiles array of ClassFile structures (represented by classes **503**), respectively.

The multi-class file is typically considerably smaller than the sum of the sizes of the individual class files that it was derived from. It can be loaded by the virtual machine during or prior to the execution of an application, instead of having to load each contained class on demand. The virtual machine is also able to take advantage of the allocation requirements information to pre-allocate all required memory for the multi-class set. This solves many of the problems associated with class loading.

Classes in a multi-class set share information between classes, and therefore are smaller. This provides the following advantages:

a) the classes take up less space on servers or storage devices;

b) the classes take less network or file transfer time to read;

c) the classes take up less memory when loaded; and

d) execution is faster, since shared constants are resolved at most once.

Multi-class sets consolidate the loading of required classes instead of loading the classes one by one. Using allocation information, only one dynamic memory allocation is needed instead of multiple allocation operations. This results in less fragmentation, less time spent in the allocator, and less waste of memory space.

Because the class files are consolidated in a single multi-class file, only a single transaction is needed to perform a network or file system search, to set up a transfer session (e.g., HTTP) and to transfer the entire set of classes. This minimizes pauses in the execution that can result from such transactions and provides for deterministic execution, with no pauses for class loading during a program run. Also, once the multi-class file is loaded and parsed, there is no need for the computer executing the program to remain connected to the source of the classes.

FIG. **6** illustrates the runtime data areas of the virtual machine when a multi-class file is processed and loaded in accordance with an embodiment of the invention. In FIG. **6**, runtime data areas **600** comprise multiple program counter registers (PC REG 1–M) and multiple stacks 1–M. One program counter register and one stack are allocated to each thread executing in the virtual machine. Each program counter register contains the address of the virtual machine instruction for the current method being executed by the respective thread. The stacks are used by the respective threads to store local variables, partial results and an operand stack.

Runtime data areas **600** further comprise heap **601**, which contains method area **602**. Heap **601** is the runtime data area from which memory for all class instances and arrays is allocated. Method area **602** is shared among all threads, and

stores class structures such as the constant pool, field and method data, and the code for methods. Within method area **602**, memory block **603**, which may or may not be contiguous, is allocated to the multi-class set of classes "S." Other regions in heap **601** may be allocated to "S" as well. Reduced constant pools 1–N, along with shared constant pool **604**, reside within block **603**.

Due to the removal of redundant constants in accordance with an embodiment of the invention, the size of block **603** required to contain reduced constant pools 1–N and shared constant pool **604** is much smaller than would be required to accommodate constant pools 1–N, were they not reduced. Also, the allocations in block **603** are much less fragmented (and may be found in contiguous memory) than the memory that would be allocated were the classes to be loaded one by one.

Thus, a method and apparatus for pre-processing and packaging class files has been described in conjunction with one or more specific embodiments. The invention is defined by the claims and their full scope of equivalents.

CHAPTER 4

The Class File Format

This chapter describes the Java Virtual Machine class file format. Each class file contains one Java type, either a class or an interface. Compliant Java Virtual Machine implementations must be capable of dealing with all class files that conform to the specification provided by this book.

A class file consists of a stream of 8-bit bytes. All 16-bit, 32-bit, and 64-bit quantities are constructed by reading in two, four, and eight consecutive 8-bit bytes, respectively. Multibyte data items are always stored in big-endian order, where the high bytes come first. In Java, this format is supported by inter-faces java.io.DataInput and java.io-.DataOutput and classes such as java.io.DataInputStream and java.io.DataOutputStream.

This chapter defines its own set of data types representing Java class file data: The types u1, u2, and u4 represent an unsigned one-, two-, or four-byte quantity, respectively. In Java, these types may be read by methods such as readUnsignedByte, readUnsignedShort, and readint of the interface java.io.DataInput.

The Java class file format is presented using pseudostructures written in a C-like structure notation. To avoid confusion with the fields of Java Virtual Machine classes and class instances, the contents of the structures describing the Java class file format are referred to as items. Unlike the fields of a C structure, successive items are stored in the Java class file sequentially, without padding or alignment.

Variable-sized tables, consisting of variable-sized items, are used in several class file structures. Although we will use C-like array syntax to refer to table items, the fact that tables are streams of varying-sized structures means that it is not possible to directly translate a table index into a byte offset into the table.

Where we refer to a data structure as an array, it is literally an array.

4.1 ClassFile

A class file contains a single ClassFile structure:

```
ClassFile {
    u4 magic;
    u2 minor_version;
    u2 major_version;
    u2 constant_pool_count;
    cp_info constant_pool[constant_pool_count-1];
    u2 access_flags;
    u2 this_class;
    u2 super_class;
    u2 interfaces_count;
    u2 interfaces[interfaces_count]
    u2 fields_count;
    field_info fields[fields_count]
    u2 methods_count;
    method_info methods[methods_count];
    u2 attributes_count;
    attribute_info attributes [attributes_count];
}
```

The items in the ClassFile structure are as follows:

magic

The magic item supplies the magic number identifying the class file format; it has the value 0xCAFEBABE.

minor_version, major_version

The values of the minor_version and major_version items are the minor and major version numbers of the compiler that produced this class file. An implementation of the Java Virtual Machine normally supports class files having a given major version number and minor version numbers 0 through some particular minor_version.

If an implementation of the Java Virtual Machine supports some range of minor version numbers and a class file of the same major version but a higher minor version is encountered, the Java Virtual Machine must not attempt to run the newer code. However, unless the major version number differs, it will be feasible to implement a new Java Virtual Machine that can run code of minor versions up to and including that of the newer code.

A Java Virtual Machine must not attempt to run code with a different major version. A change of the major version number indicates a major incompatible change, one that requires a fundamentally different Java Virtual Machine.

In Sun's Java Developer's Kit (JDK) 1.0.2 release, documented by this book, the value of major_version is 45. The value of minor_version is 3. Only Sun may define the meaning of new class file version numbers.

constant_pool_count

The value of the constantsool_count item must be greater than zero. It gives the number of entries in the constant_pool table of the class file, where the constant_pool entry at index zero is included in the count but is not present in the constant_pool table of the class file. A constant_pool index is considered valid if it is greater than zero and less than constant_pool_count.

constant_pool[ ]

The constant_pool is a table of variable-length structures (§4.4) representing various string constants, class names, field names, and other constants that are referred to within the ClassFile structure and its substructures.

The first entry of the constant_pool table, constant_pool [0], is reserved for internal use by a Java Virtual Machine implementation. That entry is not present in the class file. The first entry in the class file is constant_pool [1].

Each of the constant_pool table entries at indices 1 through constant_pool_count_1 is a variable-length structure (§4.4) whose format is indicated by its first "tag" byte.

access_flags

The value of the access_flags item is a mask of modifiers used with class and interface declarations. The access_flags modifiers are shown in Table 4.1.

Flag Name	Value	Meaning	Used By
ACC_PUBLIC	0x0001	Is public; may be accessed from outside its package.	Class, interface
ACC_FINAL	0x0010	Is final; no subclasses allowed.	Class
ACC_SUPER	0x0020	Treat superclass methods specially in invokespecial.	Class, interface
ACC_INTERFACE	0x0200	Is an interface.	Interface
ACC_ABSTRACT	0x0400	Is abstract; may not be instantiated.	Class, interface

An interface is distinguished by its ACC_INTERFACE flag being set. If ACC_INTERFACE is not set, this class file defines a class, not an interface.

Interfaces may only use flags indicated in Table 4.1 as used by interfaces. Classes may only use flags indicated in Table 4.1 as used by classes. An interface is implicitly abstract (§2.13.1); its ACC_ABSTRACT flag must be set. An interface cannot be final; its implementation could never be completed (§2.13.1) if it were, so it could not have its ACC_FINAL flag set.

The flags ACC_FINAL and ACC_ABSTRACT cannot both be set for a class; the implementation of such a class could never be completed (§2.8.2).

The setting of the ACC_SUPER flag directs the Java Virtual Machine which of two alternative semantics for its invokespecial instruction to express; it exists for backward compatibility for code compiled by Sun's older Java compilers. All new implementations of the Java Virtual Machine should implement the semantics for invokespecial documented in Chapter 6, *"Java Virtual Machine Instruction Set."* All new compilers to the Java Virtual Machine's instruction set should set the ACC_SUPER flag. Sun's older Java compilers generate ClassFile flags with ACC_SUPER unset. Sun's older Java Virtual Machine implementations ignore the flag if it is set.

All unused bits of the access_flags item, including those not assigned in Table 4.1, are reserved for future use. They should be set to zero in generated class files and should be ignored by Java Virtual Machine implementations.

this_class

The value of the this_class item must be a valid index into the constant_pool table. The constant_pool entry at that index must be a CONSTANT_Class_info (§4.4.1) structure representing the class or interface defined by this class file.

super_class

For a class, the value of the super_class item either must be zero or must be a valid index into the constant_pool table. If the value of the super_class item is nonzero, the constant_pool entry at that index must be a CONSTANT_Class_info (§4.4.1) structure representing the superclass of the class defined by this class file. Neither the superclass nor any of its superclasses may be a final class.

If the value of super_class is zero, then this class file must represent the class java.lang.Object, the only class or interface without a superclass.

For an interface, the value of super_class must always be a valid index into the constant_pool table. The constant_pool entry at that index must be a CONSTANT_Class_info structure representing the class java.lang.Object.

interfaces_count

The value of the interfaces_count item gives the number of direct superinterfaces of this class or interface type.

interfaces[ ]

Each value in the interfaces array must be a valid index into the constant_pool table. The constant_pool entry at each value of interfaces [i], where 0 £ i < interfaces_count, must be a CONSTANT_Class_info (§4.4.1) structure representing an interface which is a direct superinterface of this class or interface type, in the left-to-right order given in the source for the type.

fields_count

The value of the fields_count item gives the number of field_info structures in the fields table. The field_info (§4.5) structures represent all fields, both class variables and instance variables, declared by this class or interface type.

fields[ ]

Each value in the fields table must be a variable-length field_info(§4.5) structure giving a complete description of a field in the class or interface type. The fields table includes only those fields that are declared by this class or interface. It does not include items representing fields that are inherited from superclasses or superinterfaces.

methods_count

The value of the methods_count item gives the number of method_info structures in the methods table.

methods[ ]

Each value in the methods table must be a variable-length method_info (§4.6) structure giving a complete description of and Java Virtual Machine code for a method in the class or interface.

The method_info structures represent all methods, both instance methods and, for classes, class (static) methods, declared by this class or interface type. The methods table only includes those methods that are explicitly declared by this class. Interfaces have only the single method <clinit>, the interface initialization method (§3.8). The methods table does not include items representing methods that are inherited from superclasses or superinterfaces.

attributes_count

The value of the attributes_count item gives the number of attributes (§4.7) in the attributes table of this class.

attributes[ ]

Each value of the attributes table must be a variable-length attribute structure. A ClassFile structure can have any number of attributes (§4.7) associated with it.

The only attribute defined by this specification for the attributes table of a ClassFile structure is the SourceFile attribute (§4.7.2).

A Java Virtual Machine implementation is required to silently ignore any or all attributes in the attributes table of a ClassFile structure that it does not recognize. Attributes not defined in this specification are not allowed to affect the semantics of the class file, but only to provide additional descriptive information (§4.7.1).

4.2 Internal Form of Fully Qualified Class Names

Class names that appear in class file structures are always represented in a fully qualified form (§2.7.9). These class names are always represented as CONSTANT_Utf8_info (§4.4.7) structures, and they are referenced from those CONSTANT_NameAndType_info (§4.4.6) structures that have class names as part of their descriptor (§4.3, as well as from all CONSTANT_Class_info (§4.4.1) structures.

For historical reasons the exact syntax of fully qualified class names that appear in class file structures differs from the familiar Java fully qualified class name documented in

§2.7.9. In the internal form, the ASCII periods ('.') that normally separate the identifiers (§2.2) that make up the fully qualified name are replaced by ASCII forward slashes ('/'). For example, the normal fully qualified name of class Thread is java.lang.Thread. In the form used in descriptors in class files, a reference to the name of class Thread is implemented using a CONSTANT_Utf8_info structure representing the string "java/lang/Thread".

4.3 Descriptors

A descriptor is a string representing the type of a field or method.

4.3.1 Grammar Notation

Descriptors are specified using a grammar. This grammar is a set of productions that describe how sequences of characters can form syntactically correct descriptors of various types. Terminal symbols of the grammar are shown in bold fixed-width font. Nonterminal symbols are shown in italic type. The definition of a nonterminal is introduced by the name of the nonterminal being defined, followed by a colon. One or more alternative right-hand sides for the nonterminal then follow on succeeding lines. A nonterminal symbol on the right-hand side of a production that is followed by an asterisk (*) represents zero or more possibly different values produced from that nonterrinal, appended without any intervening space.

4.3.2 Field Descriptors

A field descriptor represents the type of a class or instance variable. It is a series of characters generated by the grammar:

FieldDescriptor:
  FieldType

ComponentType:
  FieldType

FieldType:
  BaseType
  ObjectType
  ArrayType

BaseType:
  B
  C
  D
  F
  I
  J
  S
  Z

ObjectType:
  L<classname>;

ArrayType:
  [ComponentType

The characters of BaseType, the L and; of ObjectType, and the [of ArrayType are all ASCII characters. The <classname> represents a fully qualified class name, for instance, java.lang.Thread. For historical reasons it is stored in a class file in a modified internal form (§4.2).

The meaning of the field types is as follows:

B	byte	signed byte
C	char	character
D	double	double-precision IEEE 754 float
F	float	single-precision IEEE 754 float
I	int	integer
J	long	long integer
L<classname>;	. . .	an instance of the class
S	short	signed short
Z	boolean	true or false
[	. . .	one array dimension

For example, the descriptor of an int instance variable is simply I. The descriptor of an instance variable of type Object is Ljava/lang/Object;. Note that the internal form of the fully qualified class name for class Object is used. The descriptor of an instance variable that is a multidimensional double array,

double d[ ] [ ] [ ];

is

[[[D

4.3.3 Method Descriptors

A parameter descriptor represents a parameter passed to a method:

ParameterDescriptor:
  FieldType

A method descriptor represents the parameters that the method takes and the value that it returns:

MethodDescriptor:
  (ParameterDescriptor*)ReturnDescriptor

A return descriptor represents the return value from a method. It is a series of characters generated by the grammar:

ReturnDescriptor:
  FieldType
  V

The character V indicates that the method returns no value (its return type is void). Otherwise, the descriptor indicates the type of the return value.

A valid Java method descriptor must represent 255 or fewer words of method parameters, where that limit includes the word for this in the case of instance method invocations. The limit is on the number of words of method parameters and not on the number of parameters themselves; parameters of type long and double each use two words.

For example, the method descriptor for the method

Object mymethod(int i, double d, Thread t)

is

(IDLjava/lang/Thread;)Ljava/lang/Object;

Note that internal forms of the fully qualified class names of Thread and Object are used in the method descriptor.

The method descriptor for mymethod is the same whether mymethod is static or is an instance method. Although an instance method is passed this, a reference to the current class instance, in addition to its intended parameters, that fact is not reflected in the method descriptor. (A reference to this is not passed to a static method.) The reference to this is passed implicitly by the method invocation instructions of the Java Virtual Machine used to invoke instance methods.

4.4 Constant Pool

All constant_pool table entries have the following general format:

```
cp_info {
    u1 tag;
    u1 info [ ];
}
```

Each item in the constant_pool table must begin with a 1-byte tag indicating the kind of cp_info entry. The contents of the info array varies with the value of tag. The valid tags and their values are listed in Table 4.2

Constant Type	Value
CONSTANT_Class	7
CONSTANT_Fieldref	9
CONSTANT_Methodref	10
CONSTANT_InterfaceMethodref	11
CONSTANT_String	8
CONSTANT_Integer	3
CONSTANT_Float	4
CONSTANT_Long	5
CONSTANT_Double	6
CONSTANT_NameAndType	12
CONSTANT_Utf8	1

Each tag byte must be followed by two or more bytes giving information about the specific constant. The format of the additional information varies with the tag value.

4.4.1 CONSTANT_Class

The CONSTANT_Class_info structure is used to represent a class or an interface:

```
CONSTANT_Class_info {
    u1 tag;
    u2 name_index;
}
```

The items of the CONSTANT_Class_info structure are the following:

tag

The tag item has the value CONSTANT_Class (7).

name_index

The value of the name_index item must be a valid index into the constant_pool table. The constant_pool entry at that index must be a CONSTANT_Utf8_info (§4.4.7) structure representing a valid fully qualified Java class name (§2.8.1) that has been converted to the class file's internal form (§4.2).

Because arrays are objects, the opcodes anewarray and multianewarray can reference array "classes" via CONSTANT_Class_info (§4.4.1) structures in the constant_pool table. In this case, the name of the class is the descriptor of the array type. For example, the class name representing a two-dimensional int array type;

int[ ][ ]

is

[[I

The class name representing the type array of class Thread;

Thread[ ]

is

[Ljava.lang.Thread;

A valid Java array type descriptor must have 255 or fewer array dimensions.

4.4.2 CONSTANT_Fieldref, CONSTANT_Methodref, and CONSTANT_InterfaceMethodref

Fields, methods, and interface methods are represented by similar structures:

```
CONSTANT_Fieldref_info {
    u1 tag;
    u2 class_index;
    u2 name_and_type_index;
}
```

-continued

```
CONSTANT_Methodref_info {
    u1 tag;
    u2 class_index;
    u2 name_and_type_index;
}
CONSTANT_InterfaceMethodref_info {
    u1 tag;
    u2 class_index;
    u2 name_and_type_index;
}
```

The items of these structures are as follows:

tag

The tag item of a CONSTANT_Fieldref_info structure has the value CONSTANT_Fieldref (9).

The tag item of a CONSTANT_Methodref_info structure has the value CONSTANT_Methodref (10).

The tag item of a CONSTANT_InterfaceMethodref_info structure has the value CONSTANT_InterfaceMethodref (11).

class_index

The value of the class_index item must be a valid index into the constant_pool table. The constant_pool entry at that index must be a CONSTANT_Class_info (§4.4.1) structure representing the class or interface type that contains the declaration of the field or method.

The class_index item of a CONSTANT_Fieldref_info or a CONSTANT_Methodref_info structure must be a class type, not an interface type. The class_index item of a CONSTANT_InterfaceMethodref_info structure must be an interface type that declares the given method.

name_and_type_index

The value of the name_and_type_index item must be a valid index into the constant_pool table. The constant_pool entry at that index must be a CONSTANT_NameAndType_info (§4.4.6) structure. This constant_pool entry indicates the name and descriptor of the field or method.

If the name of the method of a CONSTANT_Methodref_info or CONSTANT_InterfaceMethodref_info begins with a '<' ('u003c'), then the name must be one of the special internal methods (§3,8), either <init> or <clinit>. In this case, the method must return no value.

4.4.3 CONSTANT_String

The CONSTANT_String_info structure is used to represent constant objects of the type java.lang.String:

```
CONSTANT_String_info {
    u1 tag;
    u2 string_index;
}
```

The items of the CONSTANT_String_info structure are as follows:

tag

The tag item of the CONSTANT_String_info structure has the value CONSTANT_String (8).

string_index

The value of the string_index item must be a valid index into the constant_pool table. The constant_pool entry at that index must be a CONSTANT_Utf8_info (§4.4.3) structure representing the sequence of characters to which the java.lang.String object is to be initialized.

4.4.4 CONSTANT_Integer and CONSTANT_Float

The CONSTANT_Integer_info and CONSTANT_Float_info structures represent four-byte numeric (int and float) constants:

```
CONSTANT_Integer_info {
    u1 tag;
    u4 bytes;
}
CONSTANT_Float_info {
    u1 tag;
    u4 bytes;
}
```

The items of these structures are as follows:

tag

The tag item of the CONSTANT_Integer_info structure has the value CONSTANT_Integer (3).

The tag item of the CONSTANT_Float_info structure has the value CONSTANT_Float (4).

bytes

The bytes item of the CONSTANT_Integer_info structure contains the value of the int constant. The bytes of the value are stored in big-endian (high byte first) order.

The bytes item of the CONSTANT_Float_info structure contains the value of the float constant in IEEE 754 floating-point "single format" bit layout. The bytes of the value are stored in big-endian (high byte first) order, and are first converted into an int argument. Then:

If the argument is 0x7f800000, the float value will be positive infinity.

If the argument is 0xff800000, the float value will be negative infinity.

If the argument is in the range 0x7f800001 through 0x7fffffff or in the range 0xff800001 through 0xffffffff, the float value will be NaN.

In all other cases, let s, e, and m be three values that might be computed by

```
int s=((bytes >> 31) == 0) ? 1 : -1;
int e=((bytes >> 23) & 0xff);
int m=(e == 0) ?
  (bytes & 0x7fffff) << 1 :
  (bytes & 0x7fffff) | 0x800000;
```

Then the float value equals the result of the mathematical expression

$$s \cdot m \cdot 2^{e-150}$$

4.4.5 CONSTANT_Long and CONSTANT_Double

The CONSTANT_Long_info and CONSTANT_Double_info represent eight-byte numeric (long and double) constants:

```
CONSTANT_Long_info {
    u1 tag;
    u4 high_bytes;
    u4 low_bytes;
}
CONSTANT_Double_info {
    u1 tag;
    u4 high_bytes;
    u4 low_bytes;
}
```

All eight-byte constants take up two entries in the constant_pool table of the class file, as well as in the in-memory version of the constant pool that is constructed when a class file is read. If a CONSTANT_Long_info or CONSTANT_Double_info structure is the item in the constant_pool table at index n, then the next valid item in the pool is located at index n+2. The constant_pool index n+1 must be considered invalid and must not be used.[1]

[1]In retrospect, making eight-byte constants take two constant pool entries was a poor choice.

The items of these structures are as follows:

tag

The tag item of the CONSTANT_Long_info structure has the value CONSTANT_Long (5).

The tag item of the CONSTANT_Double_info structure has the value CONSTANT_Double (6).

high_bytes, low_bytes

The unsigned high bytes and low bytes items of the CONSTANT_Long structure together contain the value of the long constant ((long)high_bytes<<32)+low-bytes, where the bytes of each of high_bytes and low_bytes are stored in big-endian (high byte first) order.

The high_bytes and low_bytes items of the CONSTANT_Double_info structure contain the double value in IEEE 754 floating-point "double format" bit layout. The bytes of each item are stored in big-endian (high byte first) order. The high_bytes and low_bytes items are first converted into a long argument. Then:

If the argument is 0x7f80000000000000L, the double value will be positive infinity.

If the argument is 0xff80000000000000L, the double value will be negative infinity.

If the argument is in the range 0x7ff0000000000001L through 0x7fffffffffffffffL or in the range 0xfff0000000000001L through 0xffffffffffffffffL, the double value will be NaN.

In all other cases, let s, e, and m be three values that might be computed from the argument:

```
int s=((bits >> 63) == 0) ? 1 : -1;
int e=(int)((bits >> 52) & 0x7ffL);
long m=(e == 0) ?
  (bits & 0xfffffffffffffL) << 1 :
  (bits & 0xfffffffffffffL) | 0x10000000000000L;
```

Then the floating-point value equals the double value of the mathematical expression

$$s \cdot m \cdot 2^{e-1075}$$

4.4.6 CONSTANT_NameAndType

The CONSTANT_NameAndType_info structure is used to represent a field or method, without indicating which class or interface type it belongs to:

```
CONSTANT_NameAndType_info {
    u1 tag;
    u2 name_index;
    u2 descriptor_index;
}
```

The items of the CONSTANT_NameAndType_info structure are as follows:

tag

The tag item of the CONSTANT_NameAndType_info structure has the value CONSTANT_NameAndType (12).

name_index

The value of the name_index item must be a valid index into the constant_pool table. The constant_pool entry at that index must be a CONSTANT_Utf8_info (§4.4.7) structure representing a valid Java field name or method name (§2.7) stored as a simple (not fully qualified) name (§2.7.1), that is, as a Java identifier.

descriptor_index

The value of the descriptor_index item must be a valid index into the constant_pool table. The constant_pool entry at that index must be a CONSTANT_Utf 8_info (§4.4.7) structure representing a valid Java field descriptor (§4.3.2) or method descriptor (§4.3.3).

4.4.7 CONSTANT_Utf8

The CONSTANT_Utf8_info structure is used to represent constant string values.

UTF-8 strings are encoded so that character sequences that contain only non-null ASCII characters can be represented using only one byte per character, but characters of up to 16 bits can be represented. All characters in the range 'u0001' to 'u007F' are represented by a single byte:

0 bits **0–7**

The seven bits of data in the byte give the value of the character represented. The null character ('u0000') and characters in the range 'u0080', to 'u07FF' are represented by a pair of bytes x and y:

x: 1 1 0 bits **6–10** y: 1 0 bits **0–5**

The bytes represent the character with the value ((x & 0x1f) << 6)+(y & 0x3f).

Characters in the range 'u0800' to 'uFFFF' are represented by three bytes x, y, and z:

x: 1 1 1 0 bits **12–15** y: 1 0 bits **6–11** z: 1 0 bits **0–5**

The character with the value ((x & 0xf)<<12)+((y & 0x3f)<<6)+(z & 0x3f) is represented by the bytes. The bytes of multibyte characters are stored in the class file in big-endian (high byte first) order. There are two differences between this format and the "standard" UTF-8 format. First, the null byte (byte)0 is encoded using the two-byte format rather than the one-byte format, so that Java Virtual Machine UTF-8 strings never have embedded nulls. Second, only the one-byte, two-byte, and three-byte formats are used. The Java Virtual Machine does not recognize the longer UTF-8 formats.

For more information regarding the UTF-8 format, see *File System Safe UCS Transfonnation Format(FSS_UTF)*, X/Open Preliminary Specification, X/Open Company Ltd., Document Number: P316. This information also appears in ISO/IEC 10646, Annex P.

The CONSTANT_Utf8 _info structure is

```
CONSTANT_Utf8_info {
    u1 tag;
    u2 length;
    u1 bytes[length];
}
```

The items of the CONSTANT_Utf8 _info structure are the following:

tag

The tag item of the CONSTANT_Utf8_info structure has the value CONSTANT_Utf8 (1).

length

The value of the length item gives the number of bytes in the bytes array (not the length of the resulting string). The strings in the CONSTANT_Utf8_info structure are not null-terminated.

bytes[ ]

The bytes array contains the bytes of the string. No byte may have the value (byte)0 or (byte)0xf0–(byte)0xff.

4.5 Fields

Each field is described by a variable-length field_info structure. The format of this structure is

```
field_info {
    u2 access_flags
    u2 name_index;
    u2 descriptor_index;
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

The items of the field_info structure are as follows:

access_flags

The value of the access_flags item is a mask of modifiers used to describe access permission to and properties of a field. The access_flags modifiers are shown in Table 4.3.

Flag Name	Value	Meaning	Used By
ACC_PUBLIC	0x0001	Is public; may be accessed from outside its package.	Any field
ACC_PRIVATE	0x0002	Is private; usable only within the defining class.	Class field
ACC_PROTECTED	0x0004	Is protected; may be accessed within subclasses.	Class field
ACC_STATIC	0x0008	Is static.	Any field
ACC_FINAL	0x0010	Is final; no further overriding or assignment after initialization.	Any field
ACC_VOLATILE	0x0040	Is volatile; cannot be cached.	Class field
ACC_TRANSIENT	0x0080	Is transient; not written or read by a persistent object manager.	Class field

Fields of interfaces may only use flags indicated in Table 4.3 as used by any field. Fields of classes may use any of the flags in Table 4.3.

All unused bits of the access_flags item, including those not assigned in Table 4.3, are reserved for future use. They should be set to zero in generated class files and should be ignored by Java Virtual Machine implementations.

Class fields may have at most one of flags ACC_PUBLIC, ACC_PROTECTED, and ACC_PRIVATE set (§2.7.8). A class field may not have both ACC_FINAL and ACC_VOLATILE set (§2.9.1).

Each interface field is implicitly static and final (§2.13.4) and must have both its ACC_STATIC and ACC_FINAL flags set. Each interface field is implicitly public (§2.13.4) and must have its ACC_PUBLIC flag set.

name_index

The value of the name_index item must be a valid index into the constant_pool table. The constant_pool entry at that index must be a CONSTANT_Utf8_info (§4.4.7) structure which must represent a valid Java field name (§2.7) stored as a simple (not fully qualified) name (§2.7.1), that is, as a Java identifier.

descriptor_index

The value of the descriptor_index item must be a valid index into the constant_pool table. The constant_pool entry at that index must be a CONSTANT_Utf8 (§4.4.7) structure which must represent a valid Java field descriptor (§4.3.2).

attributes_count

The value of the attributes_count item indicates the number of additional attributes (§4.7) of this field.

attributes[ ]

Each value of the attributes table must be a variable-length attribute structure. A field can have any number of attributes (§4.7) associated with it.

The only attribute defined for the attributes table of a field_info structure by this specification is the ConstantValue attribute (§4.7.3).

A Java Virtual Machine implementation must recognize ConstantValue attributes in the attributes table of a field_info structure. A Java Virtual Machine implementation is required to silently ignore any or all other attributes in the attributes table that it does not recognize. Attributes not defined in this specification are not allowed to affect the semantics of the class file, but only to provide additional descriptive information (§4.7.1).

4.6 Methods

Each method, and each instance initialization method <init>, is described by a variable-length method_info structure. The structure has the following format:

```
method_info {
    u2 access_flags;
    u2 name_index;
    u2 descriptor_index;
    u2 attributes_count;
    attribute_info attributes [attributes_count];
}
```

The items of the method_info structure are as follows:

access_flags

The value of the access_flags item is a mask of modifiers used to describe access permission to and properties of a method or instance initialization method (3.8). The access_flags modifiers are shown in Table 4.4.

Flag Name	Value	Meaning	Used By
ACC_PUBLIC	0x0001	Is public; may be accessed from outside its package.	Any method
ACC_PRIVATE	0x0002	Is private; usable only within the defining class.	Class/instance method
ACC_PROTECTED	0x0004	Is protected; may be accessed within subclasses.	Class/instance method
ACC_STATIC	0x0008	Is static.	Class/instance method
ACC_FINAL	0x0010	Is final; no overriding is allowed.	Class/instance method
ACC_SYNCHRONIZED	0x0020	Is synchronized; wrap use in monitor lock.	Class/instance method
ACC_NATIVE	0x0100	Is native; implemented in a language other than Java.	Class/instance method
ACC_ABSTRACT	0x0400	Is abstract, no implementation is provided.	Any method

Methods in interfaces may only use flags indicated in Table 4.4 as used by any method. Class and instance methods (§2.10.3) may use any of the flags in Table 4.4. Instance initialization methods (§3.8) may only use ACC_PUBLIC, ACC_PROTECTED, and ACC_PRIVATE.

All unused bits of the access_flags item, including those not assigned in Table 4.4, are reserved for future use. They should be set to zero in generated class files and should be ignored by Java Virtual Machine implementations.

At most one of the flags ACC_PUBLIC, ACC_PROTECTED, and ACC_PRIVATE may be set for any method. Class and instance methods may not use ACC_ABSTRACT together with ACC_FINAL, ACC_NATIVE, or ACC_SYNCHRONIZED (that is, native and synchronized methods require an implementation). A class or instance method may not use ACC_PRIVATE with ACC_ABSTRACT (that is, a private method cannot be overridden, so such a method could never be implemented or used). A class or instance method may not use ACC_STATIC with ACC_ABSTRACT (that is, a static method is implicitly final and thus cannot be overridden, so such a method could never be implemented or used).

Class and interface initialization methods (§3.8), that is, methods named <clinit>, are called implicitly by the Java Virtual Machine; the value of their access_flags item is ignored.

Each interface method is implicitly abstract, and so must have its ACC_ABSTRACT flag set. Each interface method is implicitly public (§2.13.5), and so must have its ACC_PUBLIC flag set.

name_index

The value of the name_index item must be a valid index into the constant_pool table. The constant_pool entry at that index must be a CONSTANT_Utf8_info (§4.4.7) structure representing either one of the special internal method names (§3.8), either <init>or <clinit>, or a valid Java method name (§2.7), stored as a simple (not fully qualified) name (§2.7.1).

descriptor_index

The value of the descriptor_index item must be a valid index into the constant_pool table. The constant_pool entry at that index must be a CONSTANT_Utf8_info (§4.4.7) structure representing a valid Java method descriptor (§4.3.3).

attributes_count

The value of the attributes_count item indicates the number of additional attributes (§4.7) of this method.

attributes[ ]

Each value of the attributes table must be a variable-length attribute structure. A method can have any number of optional attributes (§4.7) associated with it.

The only attributes defined by this specification for the attributes table of a method_info structure are the Code (§4.7.4) and Exceptions (§4.7.5) attributes.

A Java Virtual Machine implementation must recognize Code (§4.7.4) and Exceptions (§4.7.5) attributes. A Java Virtual Machine implementation is required to silently ignore any or all other attributes in the attributes table of a method_info structure that it does not recognize. Attributes not defined in this specification are not allowed to affect the semantics of the class file, but only to provide additional descriptive information (§4.7.1).

4.7 Attributes

Attributes are used in the ClassFile (§4.1), field_info (§4.5), method_info (§4.6), and Code_attribute (§4.7.4) structures of the class file format. All attributes have the following general format:

```
attribute_info {
    u2 attribute_name_index;
    u4 attribute_length;
    u1 info[attribute_length];
}
```

For all attributes, the attribute_name_index must be a valid unsigned 16-bit index into the constant pool of the class. The constant_pool entry at attribute_name_index must be a CONSTANT_Utf8 (§4.4.7) string representing the name of the attribute. The value of the attribute_length item indicates the length of the subsequent information in bytes. The length does not include the initial six bytes that contain the attribute_name_index and attribute_length items.

Certain attributes are predefined as part of the class file specification. The predefined attributes are the SourceFile (4.7.2), ConstantValue (§4.7.3), Code (§4.7.4), Exceptions (§4.7.5), LineNumberTable (§4.7.6), and LocalVariableTable (§4.7.7) attributes. Within the context of their use in this specification, that is, in the attributes tables of the class file structures in which they appear, the names of these predefined attributes are reserved.

Of the predefined attributes, the Code, ConstantValue, and Exceptions attributes must be recognized and correctly read by a class file reader for correct interpretation of the class file by a Java Virtual Machine. Use of the remaining predefined attributes is optional; a class file reader may use the information they contain, and otherwise must silently ignore those attributes.

4.7.1 Defining and Naming New Attributes

Compilers for Java source code are permitted to define and emit class files containing new attributes in the attributes tables of class file structures. Java Virtual Machine implementations are permitted to recognize and use new attributes found in the attributes tables of class file structures. However, all attributes not defined as part of this Java Virtual Machine specification must not affect the semantics of class or interface types. Java Virtual Machine implementations are required to silently ignore attributes they do not recognize.

For instance, defining a new attribute to support vendor-specific debugging is permitted. Because Java Virtual Machine implementations are required to ignore attributes they do not recognize, class files intended for that particular Java Virtual Machine implementation will be usable by other implementations even if those implementations cannot make use of the additional debugging information that the class files contain.

Java Virtual Machine implementations are specifically prohibited from throwing an exception or otherwise refusing to use class files simply because of the presence of some new attribute. Of course, tools operating on class files may not run correctly if given class files that do not contain all the attributes they require.

Two attributes that are intended to be distinct, but that happen to use the same attribute name and are of the same length, will conflict on implementations that recognize either attribute. Attributes defined other than by Sun must have names chosen according to the package naming convention defined by *The Java Language Specification*. For instance, a new attribute defined by Netscape might have the name "COM.Netscape.new-attribute".

Sun may define additional attributes in future versions of this class file specification.

4.7.2 SourceFile Attribute

The SourceFile attribute is an optional fixed-length attribute in the attributes table of the ClassFile (§4.1) structure. There can be no more than one SourceFile attribute in the attributes table of a given ClassFile structure.

The SourceFile attribute has the format

```
SourceFile_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 sourcefile_index;
}
```

The items of the SourceFile_attribute structure are as follows:

attribute_name_index

The value of the attribute_name_index item must be a valid index into the constant_pool table. The constant_pool entry at that index must be a CONSTANT_Utf8_info (§4.4.7) structure representing the string "SourceFile".

attribute_length

The value of the attribute_length item of a SourceFile_attribute structure must be 2.

sourcefile_index

The value of the sourcefile_index item must be a valid index into the constant_pool table. The constant pool entry at that index must be a CONSTANT_Utf8_info (§4.4.7) structure representing the string giving the name of the source file from which this class file was compiled.

Only the name of the source file is given by the SourceFile attribute. It never represents the name of a directory containing the file or an absolute path name for the file. For instance, the SourceFile attribute might contain the file name foo.java but not the UNIX pathname /home/lindholm/foo.java.

4.7.3 ConstantValue Attribute

The Constantvalue attribute is a fixed-length attribute used in the attributes table of the field_info (§4.5) structures. A ConstantValue attribute represents the value of a constant field that must be (explicitly or implicitly) static; that is, the ACC_STATIC bit (§Table 4.3) in the flags item of the field_info structure must be set. The field is not required to be final. There can be no more than one ConstantValue attribute in the attributes table of a given field_info structure. The constant field represented by the field_info structure is assigned the value referenced by its ConstantValue attribute as part of its initialization (§2.16.4).

Every Java Virtual Machine implementation must recognize ConstantValue attributes.

The ConstantValue attribute has the format

```
ConstantValue_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 constantvalue_index;
}
```

The items of the ConstantValue_attribute structure are as follows:

attribute_name_index

The value of the attribute_name_index item must be a valid index into the constant_pool table. The constant_pool entry at that index must be a CONSTANT_Utf8_info (§4.4.7) structure representing the string "ConstantValue".

attribute_length

The value of the attribute_length item of a ConstantValue_attribute structure must be 2.

constantvalue_index

The value of the constantvalue_index item must be a valid index into the constant_pool table. The constant_pool entry at that index must give the constant value represented by this attribute.

The constant_pool entry must be of a type appropriate to the field, as shown by Table 4.5.

Field Type	Entry Type
long	CONSTANT_Long
float	CONSTANT_Float
double	CONSTANT_Double
int, short, char, byte, boolean	CONSTANT_Integer
java.lang.String	CONSTANT_String

4.7.4 Code Attribute

The Code attribute is a variable-length attribute used in the attributes table of method_info structures. A Code attribute contains the Java Virtual Machine instructions and auxiliary information for a single Java method, instance initialization method (§3.8), or class or interface initialization method (§3.8). Every Java Virtual Machine implementation must recognize Code attributes. There must be exactly one Code attribute in each method_info structure.

The Code attribute has the format

```
Code_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 max_stack
    u2 max_locals;
    u4 code_length;
    u1 code[code_length];
    u2 exception_table_length;
    {       u2 start_pc;
            u2 end_pc;
            u2 handler_pc;
            u2 catch_type;
    }       exception_table[exception_table_length];
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

The items of the Code_attribute structure are as follows:

attribute_name_index

The value of the attribute_name_index item must be a valid index into the constant_pool table. The constant_pool entry at that index must be a CONSTANT_Utf8_info (§4.4.7) structure representing the string "Code".

attribute_length

The value of the attribute_length item indicates the length of the attribute, excluding the initial six bytes.

max_stack

The value of the max_stack item gives the maximum number of words on the operand stack at any point during execution of this method.

max_locals

The value of the max_locals item gives the number of local variables used by this method, including the parameters passed to the method on invocation. The index of the first local variable is 0. The greatest local variable index for a one-word value is max_locals−1. The greatest local variable index for a two-word value is max_locals−2.

code_length

The value of the code_length item gives the number of bytes in the code array for this method. The value of code_length must be greater than zero; the code array must not be empty.

code[ ]

The code array gives the actual bytes of Java Virtual Machine code that implement the method.

When the code array is read into memory on a byte addressable machine, if the first byte of the array is aligned on a 4-byte boundary, the tableswitch and lookupswitch 32-bit offsets will be 4-byte aligned; refer to the descriptions of those instructions for more information on the consequences of code array alignment.

The detailed constraints on the contents of the code array are extensive and are given in a separate section (§4.8).

exception_table_length

The value of the exception_table_length item gives the number of entries in the exception_table table.

exception_table[ ]

Each entry in the exception_table array describes one exception handler in the code array. Each exception_table entry contains the following items:

start_pc, end_pc

The values of the two items start_pc and end_pc indicate the ranges in the code array at which the exception handler is active. The value of start_pc must be a valid index into the code array of the opcode of an instruction. The value of end_pc either must be a valid index into the code array of the opcode of an instruction, or must be equal to code_length, the length of the code array. The value of start_pc must be less than the value of end_pc.

The start_pc is inclusive and end_pc is exclusive; that is, the exception handler must be active while the program counter is within the interval [start_pc, end_pc).[2]

[2]The fact that end_pc is exclusive is an historical mistake in the Java Virtual Machine: if the Java Virtual Machine code for a method is exactly 65535 bytes long and ends with an instruction that is one byte long, then that instruction cannot be protected by an exception handler. A compiler writer can work around this bug by limiting the maximum size of the generated Java Virtual Machine code for any method, instance initialization method, or static initializer (the size of any code array) to 65534 bytes.

handler_pc

The value of the handler_pc item indicates the start of the exception handler. The value of the item must be a valid index into the code array, must be the index of the opcode of an instruction, and must be less than the value of the code_length item.

catch_type

If the value of the catch_type item is nonzero, it must be a valid index into the constant_pool table. The constant_pool entry at that index must be a CONSTANT_Class_info (§4.4.1) structure representing a class of exceptions that this exception handler is designated to catch. This class must be the class Throwable or one of its subclasses. The exception handler will be called only if the thrown exception is an instance of the given class or one of its subclasses.

If the value of the catch_type item is zero, this exception handler is called for all exceptions. This is used to implement finally (see Section 7.13, "Compiling finally").

attributes_count

The value of the attributes_count item indicates the number of attributes of the Code attribute.

attributes[ ]

Each value of the attributes table must be a variable-length attribute structure. A Code attribute can have any number of optional attributes associated with it.

Currently, the LineNumberTable (§4.7.6) and LocalVariableTable (§4.7.7) attributes, both of which contain debugging information, are defined and used with the Code attribute.

A Java Virtual Machine implementation is permitted to silently ignore any or all attributes in the attributes table of a Code attribute. Attributes not defined in this specification

are not allowed to affect the semantics of the class file, but only to provide additional descriptive information (§4.7.1).

4.7.5 Exceptions Attribute

The Exceptions attribute is a variable-length attribute used in the attributes table of a method_info (§4.6) structure. The Exceptions attribute indicates which checked exceptions a method may throw. There must be exactly one Exceptions attribute in each method_info structure.

The Exceptions attribute has the format

```
Exceptions_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 number_of_exceptions;
    u2 exception_index_table[number_of_exceptions];
}
```

The items of the Exceptions_attribute structure are as follows:

attribute_name_index

The value of the attribute_name_index item must be a valid index into the constant_pool table. The constant_pool entry at that index must be the CONSTANT_Utf8_info (§4.4.7) structure representing the string "Exceptions".

attribute_length

The value of the attribute_length item indicates the attribute length, excluding the initial six bytes.

number_of_exceptions

The value of the number_of_exceptions item indicates the number of entries in the exception_index_table.

exception_index_table[ ]

Each nonzero value in the exception_index_table array must be a valid index into the constant_pool table. For each table item, if exception index_table[i] !=0, where 0 £ i<number_of_exceptions, then the constant_pool entry at index exception_index_table[i] must be a CONSTANT_Class_info (4.4.1) structure representing a class type that this method is declared to throw.

A method should only throw an exception if at least one of the following three criteria is met:

- The exception is an instance of RuntimeException or one of its subclasses.
- The exception is an instance of Error or one of its subclasses.
- The exception is an instance of one of the exception classes specified in the exception_index_table above, or one of their subclasses.

The above requirements are not currently enforced by the Java Virtual Machine; they are only enforced at compile time. Future versions of the Java language may require more rigorous checking of throws clauses when classes are verified.

4.7.6 LineNumberTable Attribute

The LineNumberTable attribute is an optional variable-length attribute in the attributes table of a Code (§4.7.4) attribute. It may be used by debuggers to determine which part of the Java Virtual Machine code array corresponds to a given line number in the original Java source file. If LineNumberTable attributes are present in the attributes table of a given Code attribute, then they may appear in any order. Furthermore, multiple LineNumberTable attributes may together represent a given line of a Java source file; that is, LineNumberTable attributes need not be one-to-one with source lines.[3]

[3]The javac compiler in Sun's JDK 1.0.2 release can in fact generate LineNumberTable attributes which are not in line number order and which are not one-to-one with source lines. This is unfortunate, as we would prefer to specify a one-to-one, ordered mapping of LineNumberTable attributes to source lines, but must yield to backward compatibility.

The LineNumberTable attribute has the format

```
LineNumberTable_attributes {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 line_number_table_length;
    {   u2 start_pc;
        u2 line_number
    }   line_number_table[line_number_table_length];
}
```

The items of the LineNumberTable_attribute structure are as follows:

attribute_name_index

The value of the attribute_name_index item must be a valid index into the constant_pool table. The constant_pool entry at that index must be a CONSTANT_Utf8_info (§4.4.7) structure representing the string "LineNumberTable".

attribute_length

The value of the attribute_length item indicates the length of the attribute, excluding the initial six bytes.

line_number_table_length

The value of the line number_table_length item indicates the number of entries in the line_number_table array.

line_number_table[ ]

Each entry in the line_number_table array indicates that the line number in the original Java source file changes at a given point in the code array. Each entry must contain the following items:

start_pc

The value of the start_pc item must indicate the index into the code array at which the code for a new line in the original Java source file begins. The value of start_pc must be less than the value of the code_length item of the Code attribute of which this LineNumberTable is an attribute.

line_number

The value of the line_number item must give the corresponding line number in the original Java source file.

4.7.7 LocalVariableTable Attribute

The LocalVariableTable attribute is an optional variable-length attribute of a Code (§4.7.4) attribute. It may be used by debuggers to determine the value of a given local variable during the execution of a method. If LocalVariableTable attributes are present in the attributes table of a given Code attribute, then they may appear in any order. There may be no more than one LocalVariableTable attribute per local variable in the Code attribute.

The LocalVariableTable attribute has the format

```
LocalVariableTable_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 local_variable_table_length;
    {   u2 start_pc;
        u2 length;
        u2 name_index;
        u2 descriptor_index;
        u2 index;
    }   local_variable_table [local_variable_table_length];
}
```

The items of the LocalVariableTable_attribute structure are as follows:

attribute_name_index

The value of the attribute_name_index item must be a valid index into the constant_pool table. The constant_pool entry at that index must be a CONSTANT_Utf8_info (§4.4.7) structure representing the string "LocalVariableTable".

attribute_length

The value of the attribute_length item indicates the length of the attribute, excluding the initial six bytes.

local_variable_table_length

The value of the local_variable_table_length item indicates the number of entries in the local_variable_table array.

local_variable_table[ ]

Each entry in the local_variable_table array indicates a range of code array offsets within which a local variable has a value. It also indicates the index into the local variables of the current frame at which that local variable can be found. Each entry must contain the following items:

start_pc, length

The given local variable must have a value at indices into the code array in the interval [start_pc, start_pc+length], that is, between start_pc and start_pc+length inclusive. The value of start_pc must be a valid index into the code array of this Code attribute of the opcode of an instruction. The value of start_pc+length must be either a valid index into the code array of this Code attribute of the opcode of an instruction, or the first index beyond the end of that code array.

name_index, descriptor_index

The value of the name_index item must be a valid index into the constant_pool table. The constant_pool entry at that index must contain a CONSTANT_Utf8_info (§4.4.7) structure representing a valid Java local variable name stored as a simple name (§2.7.1).

The value of the descriptor_index item must be a valid index into the constant_pool table. The constant_pool entry at that index must contain a CONSTANT_Utf 8_info (§4.4.7) structure representing a valid descriptor for a Java local variable. Java local variable descriptors have the same form as field descriptors (§4.3.2).

index

The given local variable must be at index in its method's local variables. If the local variable at index is a two-word type (double or long), it occupies both index and index+1.

4.8 Constraints on Java Virtual Machine Code

The Java Virtual Machine code for a method, instance initialization method (§3.8), or class or interface initialization method (§3.8) is stored in the code array of the Code attribute of a method_info structure of a class file. This section describes the constraints associated with the contents of the Code_attribute structure.

4.8.1 Static Constraints

The static constraints on a class file are those defining the well-formedness of the file. With the exception of the static constraints on the Java Virtual Machine code of the class file, these constraints have been given in the previous section. The static constraints on the Java Virtual Machine code in a class file specify how Java Virtual Machine instructions must be laid out in the code array, and what the operands of individual instructions must be.

The static constraints on the instructions in the code array are as follows:

- The code array must not be empty, so the code_length attribute cannot have the value 0.
- The opcode of the first instruction in the code array begins at index 0.
- Only instances of the instructions documented in (§6.4) may appear in the code array. Instances of instructions using the reserved opcodes (§6.2), the _quick opcodes documented in Chapter 9, "An Optimization," or any opcodes not documented in this specification may not appear in the code array.
- For each instruction in the code array except the last, the index of the opcode of the next instruction equals the index of the opcode of the current instruction plus the length of that instruction, including all its operands. The wide instruction is treated like any other instruction for these purposes; the opcode specifying the operation that a wide instruction is to modify is treated as one of the operands of that wide instruction. That opcode must never be directly reachable by the computation.
- The last byte of the last instruction in the code array must be the byte at index code_length_1.

The static constraints on the operands of instructions in the code array are as follows:

- The target of each jump and branch instruction (jsr, jsr_w, goto, goto_w, ifeq, ifne, iflt, ifge, ifgt, ifle, ifnull, ifnonnull, if_icmpeq, if_icmpne, if_icmplt, if_icmpge, if_icmpgt, if_icmple, if_acmpeq, if_acmpne) must be the opcode of an instruction within this method. The target of a jump or branch instruction must never be the opcode used to specify the operation to be modified by a wide instruction; a jump or branch target may be the wide instruction itself.
- Each target, including the default, of each tableswitch instruction must be the opcode of an instruction within this method. Each tableswitch instruction must have a number of entries in its jump table that is consistent with its low and high jump table operands, and its low value must be less than or equal to its high value. No target of a tableswitch instruction may be the opcode used to specify the operation to be modified by a wide instruction; a tableswitch target may be a wide instruction itself.
- Each target, including the default, of each lookupswitch instruction must be the opcode of an instruction within this method. Each lookupswitch instruction must have a number of match-offset pairs that is consistent with its npairs operand. The match-offset pairs must be sorted in increasing numerical order by signed match value. No target of a lookupswitch instruction may be the opcode used to specify the operation to be modified by a wide instruction; a lookupswitch target may be a wide instruction itself
- The operand of each ldc and ldc_w instruction must be a valid index into the constant_pool table. The constant pool entry referenced by that index must be of type CONSTANT_Integer, CONSTANT_Float, or CONSTANT_String.
- The operand of each ldc2_w instruction must be a valid index into the constant_pool table. The constant pool entry referenced by that index must be of type CONSTANT_Long or CONSTANT_double. In addition, the subsequent constant pool index must also be a valid index into the constant pool, and the constant pool entry at that index must not be used.
- The operand of each getfield, putfield, getstatic, and putstatic instruction must be a valid index into the constant_pool table. The constant pool entry referenced by that index must be of type CONSTANT_Fieldref.
- The index operand of each invokevirtual, invokespecial, and invokestatic instruction must be a valid index into the constant_pool table. The constant pool entry referenced by that index must be of type CONSTANT_Methodref.

Only the invokespecial instruction is allowed to invoke the method <init>, the instance initialization method (§3.8). No other method whose name begins with the character '<' ('u003c') may be called by the method invocation instructions. In particular, the class initialization method <clinit> is never called explicitly from Java Virtual Machine instructions, but only implicitly by the Java Virtual Machine itself.

The index operand of each invokeinterface instruction must be a valid index into the constant_pool table. The constant pool entry referenced by that index must be of type CONSTANT_InterfaceMethodref. The value of the nargs operand of each invokeinterface instruction must be the same as the number of argument words implied by the descriptor of the CONSTANT_NameAndType_info structure referenced by the CONSTANT_InterfaceMethodref constant pool entry. The fourth operand byte of each invokeinterface instruction must have the value zero.

The index operand of each instanceof, checkcast, new, anewarray, and multi-anewarray instruction must be a valid index into the constant_pool table. The constant pool entry referenced by that index must be of type CONSTANT_Class.

No anewarray instruction may be used to create an array of more than 255 dimensions.

No new instruction may reference a CONSTANT_Class constant_pool table entry representing an array class. The new instruction cannot be used to create an array. The new instruction also cannot be used to create an interface or an instance of an abstract class, but those checks are performed at link time.

A multianewarray instruction must only be used to create an array of a type that has at least as many dimensions as the value of its dimensions operand. That is, while a multianewarray instruction is not required to create all of the dimensions of the array type referenced by its CONSTANT_Class operand, it must not attempt to create more dimensions than are in the array type. The dimensions operand of each multianewarray instruction must not be zero.

The atype operand of each newarray instruction must take one of the values T_BOOLEAN (4), T_CHAR (5), T_FLOAT (§6), T_DOUBLE (7), T_BYTE (8), T_SHORT (9), T_INT (10), or T_LONG (11).

The index operand of each iload, fload, aload, istore, fstore, astore, wide, iinc, and ret instruction must be a natural number no greater than max_locals−1.

The implicit index of each iload_<n>, fload_<n>, $aload_{13}$ <n>, istore_<n>, fstore_<n>, and astore_<n> instruction must be no greater than the value of max_locals−1.

The index operand of each lload, dload, lstore, and dstore instruction must be no greater than the value of max_locals−2.

The implicit index of each lload_<n>, dload_<n>, lstore_<n>, and dstore_<n> instruction must be no greater than the value of max_locals−2.

4.8.2 Structural Constraints

The structural constraints on the code array specify constraints on relationships between Java Virtual Machine instructions. The structural constraints are as follows:

Each instruction must only be executed with the appropriate type and number of arguments in the operand stack and local variables, regardless of the execution path that leads to its invocation. An instruction operating on values of type int is also permitted to operate on values of type byte, char, and short. (As noted in §3.11.1, the Java Virtual Machine internally converts values of types byte, char, and short to type int.)

Where an instruction can be executed along several different execution paths, the operand stack must have the same size prior to the execution of the instruction, regardless of the path taken.

At no point during execution can the order of the words of a two-word type (long or double) be reversed or split up. At no point can the words of a two-word type be operated on individually.

No local variable (or local variable pair, in the case of a two-word type) can be accessed before it is assigned a value.

At no point during execution can the operand stack grow to contain more than max_stack words.

At no point during execution can more words be popped from the operand stack than it contains.

Each invokespecial instruction must name only an instance initialization method <init>, a method in this, a private method, or a method in a superclass of this.

When the instance initialization method <init>is invoked, an uninitialized class instance must be in an appropriate position on the operand stack. The <init> method must never be invoked on an initialized class instance.

When any instance method is invoked, or when any instance variable is accessed, the class instance that contains the instance method or instance variable must already be initialized.

There must never be an uninitialized class instance on the operand stack or in a local variable when any backwards branch is taken. There must never be an uninitialized class instance in a local variable in code protected by an exception handler or a finally clause. However, an uninitialized class instance may be on the operand stack in code protected by an exception handler or a finally clause. When an exception is thrown, the contents of the operand stack are discarded.

Each instance initialization method (§3.8), except for the instance initialization method derived from the constructor of class Object, must call either another instance initialization method of this or an instance initialization method of its immediate superclass super before its instance members are accessed. However, this is not necessary in the case of class Object, which does not have a superclass (§2.4.6).

The arguments to each method invocation must be method invocation compatible (§2.6.7) with the method descriptor (§4.3.3).

An abstract method must never be invoked.

Each return instruction must match its method's return type. If the method returns a byte, char, short, or int, only the ireturn instruction may be used. If the method returns a float, long, or double, only an freturn, lreturn, or dreturn instruction, respectively, may be used. If the method returns a reference type, it must do so using an areturn instruction, and the returned value must be assignment compatible (§2.6.6) with the return descriptor (§4.3.3) of the method. All instance initialization methods, static initializers, and methods declared to return void must only use the return instruction.

If getfield or putfield is used to access a protected field of a superclass, then the type of the class instance being

accessed must be the same as or a subclass of the current class. If invokevirtual is used to access a protected method of a superclass, then the type of the class instance being accessed must be the same as or a subclass of the current class.

The type of every class instance loaded from or stored into by a getfield or putfield instruction must be an instance of the class type or a subclass of the class type.

The type of every value stored by a putfield or putstatic instruction must be compatible with the descriptor of the field (§4.3.2) of the class instance or class being stored into. If the descriptor type is byte, char, short, or int, then the value must be an int. If the descriptor type is float, long, or double, then the value must be a float, long, or double, respectively. If the descriptor type is a reference type, then the value must be of a type that is assignment compatible (§2.6.6) with the descriptor type.

The type of every value stored into an array of type reference by an aastore instruction must be assignment compatible (§2.6.6) with the component type of the array.

Each athrow instruction must only throw values that are instances of class Throwable or of subclasses of Throwable.

Execution never falls off the bottom of the code array.

No return address (a value of type returnAddress) may be loaded from a local variable.

The instruction following each jsr or jsr_w instruction only may be returned to by a single ret instruction.

No jsr or jsr_w instruction may be used to recursively call a subroutine if that subroutine is already present in the subroutine call chain. (Subroutines can be nested when using try-finally constructs from within a finally clause. For more information on Java Virtual Machine subroutines, see §4.9.6)

Each instance of type returnaddress can be returned to at most once. If a ret instruction returns to a point in the subroutine call chain above the ret instruction corresponding to a given instance of type returnAddress, then that instance can never be used as a return address.

4.9 Verification of Class Files

Even though Sun's Java compiler attempts to produce only class files that satisfy all the static constraints in the previous sections, the Java Virtual Machine has no guarantee that any file it is asked to load was generated by that compiler, or is properly formed. Applications such as Sun's HotJava World Wide Web browser do not download source code which they then compile; these applications download already-compiled class files. The HotJava browser needs to determine whether the class file was produced by a trustworthy Java compiler or by an adversary attempting to exploit the interpreter.

An additional problem with compile-time checking is version skew. A user may have successfully compiled a class, say PurchaseStockOptions, to be a subclass of TradingClass. But the definition of TradingClass might have changed in a way that is not compatible with preexisting binaries since the time the class was compiled. Methods might have been deleted, or had their return types or modifiers changed. Fields might have changed types or changed from instance variables to class variables. The access modifiers of a method or variable may have changed from public to private. For a discussion of these issues, see Chapter 13, "Binary Compatibility," in *The Java Language Specification.*

Because of these potential problems, the Java Virtual Machine needs to verify for itself that the desired constraints hold on the class files it attempts to incorporate. A well-written Java Virtual Machine emulator could reject poorly formed instructions when a class file is loaded. Other constraints could be checked at run time. For example, a Java Virtual Machine implementation could tag runtime data and have each instruction check that its operands are of the right type.

Instead, Sun's Java Virtual Machine implementation verifies that each class file it considers untrustworthy satisfies the necessary constraints at linking time (§2.16.3). Structural constraints on the Java Virtual Machine code are checked using a simple theorem prover.

Linking-time verification enhances the performance of the interpreter. Expensive checks that would otherwise have to be performed to verify constraints at run time for each interpreted instruction can be eliminated. The Java Virtual Machine can assume that these checks have already been performed. For example, the Java Virtual Machine will already know the following:

There are no operand stack overflows or underflows.

All local variable uses and stores are valid.

The arguments to all the Java Virtual Machine instructions are of valid types.

Sun's class file verifier is independent of any Java compiler. It should certify all code generated by Sun's current Java compiler; it should also certify code that other compilers can generate, as well as code that the current compiler could not possibly generate. Any class file that satisfies the structural criteria and static constraints will be certified by the verifier.

The class file verifier is also independent of the Java language. Other languages can be compiled into the class format, but will only pass verification if they satisfy the same constraints as a class file compiled from Java source.

4.9.1 The Verification Process

The class file verifier operates in four passes:

Pass 1: When a prospective class file is loaded (§2.16.2) by the Java Virtual Machine, the Java Virtual Machine first ensures that the file has the basic format of a Java class file. The first four bytes must contain the right magic number. All recognized attributes must be of the proper length. The class file must not be truncated or have extra bytes at the end. The constant pool must not contain any superficially unrecognizable information.

While class file verification properly occurs during class linking (§2.16.3), this check for basic class file integrity is necessary for any interpretation of the class file contents and can be considered to be logically part of the verification process.

Pass 2: When the class file is linked, the verifier performs all additional verification that can be done without looking at the code array of the Code attribute (§4.7.4). The checks performed by this pass include the following:

Ensuring that final classes are not subclassed, and that final methods are not overridden.

Checking that every class (except Object) has a superclass.

Ensuring that the constant pool satisfies the documented static constraints; for example, class references in the constant pool must contain a field that points to a CONSTANT_Utf8 string reference in the constant pool.

Checking that all field references and method references in the constant pool have valid names, valid classes, and a valid type descriptor.

Note that when it looks at field and method references, this pass does not check to make sure that the given field or method actually exists in the given class; nor does it check that the type descriptors given refer to real classes. It only checks that these items are well formed. More detailed checking is delayed until passes 3 and 4.

Pass 3: Still during linking, the verifier checks the code array of the Code attribute for each method of the class file by performing data-flow analysis on each method. The verifier ensures that at any given point in the program, no matter what code path is taken to reach that point:

- The operand stack is always the same size and contains the same types of objects.
- No local variable is accessed unless it is known to contain a value of an appropriate type.
- Methods are invoked with the appropriate arguments.
- Fields are assigned only using values of appropriate types.
- All opcodes have appropriate type arguments on the operand stack and in the local variables.

For further information on this pass, see Section 4.9.2, "The Bytecode Verifier."

Pass 4: For efficiency reasons, certain tests that could in principle be performed in Pass 3 are delayed until the first time the code for the method is actually invoked. In so doing, Pass 3 of the verifier avoids loading class files unless it has to.

For example, if a method invokes another method that returns an instance of class A, and that instance is only assigned to a field of the same type, the verifier does not bother to check if the class A actually exists. However, if it is assigned to a field of the type B, the definitions of both A and B must be loaded in to ensure that A is a subclass of B.

Pass 4 is a virtual pass whose checking is done by the appropriate Java Virtual Machine instructions. The first time an instruction that references a type is executed, the executing instruction does the following:

- Loads in the definition of the referenced type if it has not already been loaded.
- Checks that the currently executing type is allowed to reference the type.
- Initializes the class, if this has not already been done.

The first time an instruction invokes a method, or accesses or modifies a field, the executing instruction does the following:

- Ensures that the referenced method or field exists in the given class.
- Checks that the referenced method or field has the indicated descriptor.
- Checks that the currently executing method has access to the referenced method or field.

The Java Virtual Machine does not have to check the type of the object on the operand stack. That check has already been done by Pass 3. Errors that are detected in Pass 4 cause instances of subclasses of LinkageError to be thrown.

A Java Virtual Machine is allowed to perform any or all of the Pass 4 steps, except for class or interface initialization, as part of Pass 3; see 2.16. 1. "Virtual Machine Start-up" for an example and more discussion.

In Sun's Java Virtual Machine implementation, after the verification has been performed, the instruction in the Java Virtual Machine code is replaced with an alternative form of the instruction (see Chapter 9, "An Optimization"). For example, the opcode new is replaced with new_quick. This alternative instruction indicates that the verification needed by this instruction has taken place and does not need to be performed again. Subsequent invocations of the method will thus be faster. It is illegal for these alternative instruction forms to appear in class files, and they should never be encountered by the verifier.

4.9.2 The Bytecode Verifier

As indicated earlier, Pass 3 of the verification process is the most complex of the four passes of class file verification. This section looks at the verification of Java Virtual Machine code in more detail.

The code for each method is verified independently. First, the bytes that make up the code are broken up into a sequence of instructions, and the index into the code array of the start of each instruction is placed in an array. The verifier then goes through the code a second time and parses the instructions. During this pass a data structure is built to hold information about each Java Virtual Machine instruction in the method. The operands, if any, of each instruction are checked to make sure they are valid. For instance:

- Branches must be within the bounds of the code array for the method.
- The targets of all control-flow instructions are each the start of an instruction. In the case of a wide instruction, the wide opcode is considered the start of the instruction, and the opcode giving the operation modified by that wide instruction is not considered to start an instruction. Branches into the middle of an instruction are disallowed.
- No instruction can access or modify a local variable at an index greater than the number of local variables that its method indicates it uses.
- All references to the constant pool must be to an entry of the appropriate type. For example: the instruction ldc can only be used for data of type int or float, or for instances of class String; the instruction getfield must reference a field.
- The code does not end in the middle of an instruction.
- Execution cannot fall off the end of the code.
- For each exception handler, the starting and ending point of code protected by the handler must be at the beginning of an instruction. The starting point must be before the ending point. The exception handler code must start at a valid instruction, and it may not start at an opcode being modified by the wide instruction.

For each instruction of the method, the verifier records the contents of the operand stack and the contents of the local variables prior to the execution of that instruction. For the operand stack, it needs to know the stack height and the type of each value on it. For each local variable, it needs to know either the type of the contents of that local variable, or that the local variable contains an unusable or unknown value (it might be uninitialized). The bytecode verifier does not need to distinguish between the integral types (e.g., byte, short, char) when determining the value types on the operand stack.

Next, a data-flow analyzer is initialized. For the first instruction of the method, the local variables which represent parameters initially contain values of the types indicated by the method's type descriptor; the operand stack is empty. All other local variables contain an illegal value. For the other instructions, which have not been examined yet, no information is available regarding the operand stack or local variables.

Finally, the data-flow analyzer is run. For each instruction, a "changed" bit indicates whether this instruction needs to be looked at. Initially, the "changed" bit is only set for the first instruction. The data-flow analyzer executes the following loop:

1. Select a virtual machine instruction whose "changed" bit is set. If no instruction remains whose "changed" bit is set, the method has successfully been verified. Otherwise, turn off the "changed" bit of the selected instruction.
2. Model the effect of the instruction on the operand stack and local variables:
   If the instruction uses values from the operand stack, ensure that there are a sufficient number of values on the stack and that the top values on the stack are of an appropriate type. Otherwise, verification fails.
   If the instruction uses a local variable, ensure that the specified local variable contains a value of the appropriate type. Otherwise, verification fails.
   If the instruction pushes values onto the operand stack, ensure that there is sufficient room on the operand stack for the new values. Add the indicated types to the top of the modeled operand stack.
   If the instruction modifies a local variable, record that the local variable now contains the new type.
3. Determine the instructions that can follow the current instruction. Successor instructions can be one of the following:
   The next instruction, if the current instruction is not an unconditional control transfer instruction (for instance goto, return or athrow). Verification fails if it is possible to "fall off" the last instruction of the method.
   The target(s) of a conditional or unconditional branch or switch.
   Any exception handlers for this instruction.
4. Merge the state of the operand stack and local variables at the end of the execution of the current instruction into each of the successor instructions. In the special case of control transfer to an exception handler, the operand stack is set to contain a single object of the exception type indicated by the exception handler information.
   If this is the first time the successor instruction has been visited, record that the operand stack and local variables values calculated in steps 2 and 3 are the state of the operand stack and local variables prior to executing the successor instruction. Set the "changed" bit for the successor instruction.
   If the successor instruction has been seen before, merge the operand stack and local variable values calculated in steps 2 and 3 into the values already there. Set the "changed" bit if there is any modification to the values.
5. Continue at step 1.

To merge two operand stacks, the number of values on each stack must be identical. The types of values on the stacks must also be identical, except that differently typed reference values may appear at corresponding places on the two stacks. In this case, the merged operand stack contains a reference to an instance of the first common superclass or common superinterface of the two types. Such a reference type always exists because the type Object is a supertype of all class and interface types. If the operand stacks cannot be merged, verification of the method fails.

To merge two local variable states, corresponding pairs of local variables are compared. If the two types are not identical, then unless both contain reference values, the verifier records that the local variable contains an unusable value. If both of the pair of local variables contain reference values, the merged state contains a reference to an instance of the first common superclass of the two types.

If the data-flow analyzer runs on a method without reporting a verification failure, then the method has been successfully verified by Pass 3 of the class file verifier.

Certain instructions and data types complicate the data-flow analyzer. We now examine each of these in more detail.

4.9.3 Long Integers and Doubles

Values of the long and double types each take two consecutive words on the operand stack and in the local variables.

Whenever a long or double is moved into a local variable, the subsequent local variable is marked as containing the second half of a long or double. This special value indicates that all references to the long or double must be through the index of the lower-numbered local variable.

Whenever any value is moved to a local variable, the preceding local variable is examined to see if it contains the first word of a long or a double. If so, that preceding local variable is changed to indicate that it now contains an unusable value. Since half of the long or double has been overwritten, the other half must no longer be used.

Dealing with 64-bit quantities on the operand stack is simpler; the verifier treats them as single units on the stack. For example, the verification code for the dadd opcode (add two double values) checks that the top two items on the stack are both of type double. When calculating operand stack length, values of type long and double have length two.

Untyped instructions that manipulate the operand stack must treat values of type double and long as atomic. For example, the verifier reports a failure if the top value on the stack is a double and it encounters an instruction such as pop or dup. The instructions pop2 or dup2 must be used instead.

4.9.4 Instance Initialization Methods and Newly Created Objects

Creating a new class instance is a multistep process. The Java statement

```
. . .
new myclass(i, j, k);
. . .
```

can be implemented by the following:

```
. . .
new       #1      // Allocate uninitialized space for myClass
dup               // Duplicate object on the operand stack
iload_1           // Push i
iload_2           // Push j
iload_3           // Push k
invokespecial myClass.<init>      // Initialize object
. . .
```

This instruction sequence leaves the newly created and initialized object on top of the operand stack. (More examples of compiling Java code to the instruction set of the Java Virtual Machine are given in Chapter 7, "Compiling for the Java Virtual Machine.")

The instance initialization method <init>for class myClass sees the new uninitialized object as its this argument in local variable 0. It must either invoke an alternative instance initialization method for class myClass or invoke the initialization method of a superclass on the this object before it is allowed to do anything else with this.

When doing dataflow analysis on instance methods, the verifier initializes local variable 0 to contain an object of the current class, or, for instance initialization methods, local variable 0 contains a special type indicating an uninitialized object. After an appropriate initialization method is invoked (from the current class or the current superclass) on this object, all occurrences of this special type on the verifier's

model of the operand stack and in the local variables are replaced by the current class type. The verifier rejects code that uses the new object before it has been initialized or that initializes the object twice. In addition, it ensures that every normal return of the method has either invoked an initialization method in the class of this method or in the direct superclass.

Similarly, a special type is created and pushed on the verifier's model of the operand stack as the result of the Java Virtual Machine instruction new. The special type indicates the instruction by which the class instance was created and the type of the uninitialized class instance created. When an initialization method is invoked on that class instance, all occurrences of the special type are replaced by the intended type of the class instance. This change in type may propagate to subsequent instructions as the dataflow analysis proceeds.

The instruction number needs to be stored as part of the special type, as there may be multiple not-yet-initialized instances of a class in existence on the operand stack at one time. For example, the Java Virtual Machine instruction sequence that implements

new InputStream(new Foo( ), new InputStream("foo"))

may have two uninitialized instances of InputStream on the operand stack at once. When an initialization method is invoked on a class instance, only those occurrences of the special type on the operand stack or in the registers that are the same object as the class instance are replaced.

A valid instruction sequence must not have an uninitialized object on the operand stack or in a local variable during a backwards branch, or in a local variable in code protected by an exception handler or a finally clause. Otherwise, a devious piece of code might fool the verifier into thinking it had initialized a class instance when it had, in fact, initialized a class instance created in a previous pass through the loop.

4.9.5 Exception Handlers

Java Virtual Machine code produced from Sun's Java compiler always generates exception handlers such that:

- The ranges of instructions protected by two different exception handlers always are either completely disjoint, or else one is a subrange of the other. There is never a partial overlap of ranges.
- The handler for an exception will never be inside the code that is being protected.
- The only entry to an exception handler is through an exception. It is impossible to fall through or "goto" the exception handler.

These restrictions are not enforced by the class file verifier since they do not pose a threat to the integrity of the Java Virtual Machine. As long as every nonexceptional path to the exception handler causes there to be a single object on the operand stack, and as long as all other criteria of the verifier are met, the verifier will pass the code.

4.9.6 Exceptions and Finally

Given the fragment of Java code

```
. . .
try {
    startFaucet ( );
    waterLawn ( );
} finally {
    stopFaucet ( );
}
. . .
```

the Java language guarantees that stopFaucet is invoked (the faucet is turned off) whether we finish watering the lawn or whether an exception occurs while starting the faucet or watering the lawn. That is, the finally clause is guaranteed to be executed whether its try clause completes normally, or completes abruptly by throwing an exception.

To implement the try-finally construct, the Java compiler uses the exception-handling facilities together with two special instructions jsr ("jump to subroutine") and ret ("return from subroutine"). The finally clause is compiled as a subroutine within the Java Virtual Machine code for its method, much like the code for an exception handler. When a jsr instruction that invokes the subroutine is executed, it pushes its return address, the address of the instruction after the jsr that is being executed, onto the operand stack as a value of type returnAddress. The code for the subroutine stores the return address in a local variable. At the end of the subroutine, a ret instruction fetches the return address from the local variable and transfers control to the instruction at the return address.

Control can be transferred to the finally clause (the finally subroutine can be invoked) in several different ways. If the try clause completes normally, the finally subroutine is invoked via a jsr instruction before evaluating the next Java expression. A break or continue inside the try clause that transfers control outside the try clause executes a jsr to the code for the finally clause first. If the try clause executes a return, the compiled code does the following:

1. Saves the return value (if any) in a local variable.
2. Executes a jsr to the code for the finally clause.
3. Upon return from the finally clause, returns the value saved in the local variable.

The compiler sets up a special exception handler which catches any exception thrown by the try clause. If an exception is thrown in the try clause, this exception handler does the following:

1. Saves the exception in a local variable.
2. Executes a jsr to the finally clause.
3. Upon return from the finally clause, rethrows the exception.

For more information about the implementation of Java's try-finally construct, see Section 7.13. "Compiling finally."

The code for the finally clause presents a special problem to the verifier. Usually, if a particular instruction can be reached via multiple paths and a particular local variable contains incompatible values through those multiple paths, then the local variable becomes unusable. However, a finally clause might be called from several different places, yielding several different circumstances:

The invocation from the exception handler may have a certain local variable that contains an exception.

The invocation to implement return may have some local variable that contains the return value.

The invocation from the bottom of the try clause may have an indeterminate value in that same local variable.

The code for the finally clause itself might pass verification, but after updating all the successors of the ret instruction, the verifier would note that the local variable that the exception handler expects to hold an exception, or that the return code expects to hold a return value, now contains an indeterminate value.

Verifying code that contains a finally clause is complicated. The basic idea is the following:

Each instruction keeps track of the list of jsr targets needed to reach that instruction. For most code, this list is empty. For instructions inside code for the finally clause, it is of length one. For multiply nested finally code (extremely rare!), it may be longer than one.

For each instruction and each isr needed to reach that instruction, a bit vector is maintained of all local variables accessed or modified since the execution of the jsr instruction.

When executing the ret instruction, which implements a return from a subroutine, there must be only one possible subroutine from which the instruction can be returning. Two different subroutines cannot "merge" their execution to a single ret instruction.

To perform the data-flow analysis on a ret instruction, a special procedure is used. Since the verifier knows the subroutine from which the instruction must be returning, it can find all the jsr instructions that call the subroutine and merge the state of the operand stack and local variables at the time of the ret instruction into the operand stack and local variables of the instructions following the jsr. Merging uses a special set of values for the local variables:

For any local variable for which the bit vector (constructed above) indicates that the subroutine has accessed or modified, use the type of the local variable at the time of the ret.

For other local variables, use the type of the local variable before the jsr instruction.

4.10 Limitations of the Java Virtual Machine and Class File Format

The following limitations in the Java Virtual Machine are imposed by this version of the Java Virtual Machine specification:

The per-class constant pool is limited to 65535 entries by the 16-bit constant_pool_count field of the ClassFile structure (§4.1). This acts as an internal limit on the total complexity of a single class.

The amount of code per method is limited to 65535 bytes by the sizes of the indices in the exception_table of the Code attribute (§4.7.4), in the LineNumberTable attribute (§4.7.6), and in the LocalVariableTable attribute (§4.7.7).

The number of local variables in a method is limited to 65535 by the two-byte index operand of many Java Virtual Machine instructions and the size of the max_locals item of the ClassFile structure (§4.1). (Recall that values of type long and double are considered to occupy two local variables.)

The number of fields of a class is limited to 65535 by the size of the fields_count item of the ClassFile structure (§4.1).

The number of methods of a class is limited to 65535 by the size of the methods_count item of the ClassFile structure (§4.1).

The size of an operand stack is limited to 65535 words by the max_stack field of the Code_attribute structure (§4.7.4).

The number of dimensions in an array is limited to 255 by the size of the dimensions opcode of the mulitianewarray instruction, and by the constraints imposed on the multianewarray, anewarray, and newarray instructions by §4.8.2.

A valid Java method descriptor (§4.3.3) must require 255 or fewer words of method arguments, where that limit includes the word for this in the case of instance method invocations. Note that the limit is on the number of words of method arguments, and not on number of arguments themselves. Arguments of type long and double are two words long; arguments of all other types are one word long.

CHAPTER 5

Constant Pool Resolution

Java classes and interfaces are dynamically loaded (§2.16.2), linked (§2.16.3), and initialized A(§2.16.4). Loading is the process of finding the binary form of a class or interface type with a particular name and constructing, from that binary form, a Class object to represent the class or interface. Linking is the process of taking a binary form of a class or interface type and combining it into the runtime state of the Java Virtual Machine so that it can be executed. Initialization of a class consists of executing its static initializers and the initializers for static fields declared in the class.

The Java Virtual Machine performs most aspects of these procedures through operations on a constant pool (§4.4), a per-type runtime data structure that serves many of the purposes of the symbol table of a conventional language. For example, Java Virtual Machine instructions that might otherwise have been designed to take immediate numeric or string operands instead fetch their operands from the constant pool. Classes, methods, and fields, whether referenced from Java Virtual Machine instructions or from other constant pool entries, are named using the constant pool.

A Java compiler does not presume to know the way in which a Java Virtual Machine lays out classes, interfaces, class instances, or arrays. References in the constant pool are always initially symbolic. At run time, the symbolic representation of the reference in the constant pool is used to work out the actual location of the referenced entity. The process of dynamically determining concrete values from symbolic references in the constant pool is known as constant pool resolution. Constant pool resolution may involve loading one or more classes or interfaces, linking several types, and initializing types. There are several kinds of constant pool entries, and the details of resolution differ with the kind of entry to be resolved.

Individual Java Virtual Machine instructions that reference entities in the constant pool are responsible for resolving the entities they reference. Constant pool entries that are referenced from other constant pool entries are resolved when the referring entry is resolved.

A given constant pool entry may be referred to from any number of Java Virtual Machine instructions or other constant pool entries; thus, constant pool resolution can be attempted on a constant pool entry that is already resolved. An attempt to resolve a constant pool entry that has already been successfully resolved always succeeds trivially, and always results in the same entity produced by the initial resolution of that entry.

Constant pool resolution is normally initiated by the execution of a Java Virtual Machine instruction that references the constant pool. Rather than give the full description of the resolution process performed by Java Virtual Machine instructions in their individual descriptions, we will use this chapter to summarize the constant pool resolution process. We will specify the errors that must be detected when resolving each kind of constant pool entry, the order in which those errors must be responded to, and the errors thrown in response.

When referenced from the context of certain Java Virtual Machine instructions, additional constraints are put on linking operations. For instance, the getfield instruction requires not only that the constant pool entry for the

field it references can be successfully resolved, but also that the resolved field is not a class (static) field. If it is a class field, an exception must be thrown. Linking exceptions that are specific to the execution of a particular Java Virtual Machine instruction are given in the description of that instruction and are not covered in this general discussion of constant pool resolution. Note that such exceptions, although described as part of the execution of Java Virtual Machine instructions rather than constant pool resolution, are still properly considered failure of the linking phase of Java Virtual Machine execution.

The Java Virtual Machine specification documents and orders all exceptions that can arise as a result of constant pool resolution. It does not mandate how they should be detected, only that they must be. In addition, as mentioned in §6.3, any of the virtual machine errors listed as subclasses of VirtualMachineError may be thrown at any time during constant pool resolution.

5.1 Class and Interface Resolution

A constant pool entry tagged as CONSTANT_Class (§4.4.1) represents a class or interface. Various Java Virtual Machine instructions reference CONSTANT_Class entries in the constant pool of the class that is current upon their execution (§3.6). Several other kinds of constant pool entries (§4.4.2) reference CONSTANT_Class entries and cause those class or interface references to be resolved when the referencing entries are resolved. For instance, before a method reference (a CONSTANT_Methodref constant pool entry) can be resolved, the reference it makes to the class of the method (via the class_index item of the constant pool entry) must first be resolved.

If a class or interface has not been resolved already, the details of the resolution process depend on what kind of entity is represented by the CONSTANT_Class entry being resolved. Array classes are handled differently from non-array classes and from interfaces. Details of the resolution process also depend on whether the reference prompting the resolution of this class or interface is from a class or interface that was loaded using a class loader (§2.16.2).

The name_index item of a CONSTANT_Class constant pool entry is a reference to a CONSTANT_Utf8 constant pool entry (§4.4.7) for a UTF-8 string that represents the fully qualified name (§2.7.9) of the class or interface to be resolved. What kind of entity is represented by a CONSTANT_Class constant pool entry, and how to resolve that entry, is determined as follows:

- If the first character of the fully qualified name of the constant pool entry to be resolved is not a left bracket ("["), then the entry is a reference to a non-array class or to an interface.
- If the current class (§3.6) has not been loaded by a class loader, then "normal" class resolution is used (§5.1.1).
- If the current class has been loaded by a class loader, then application-defined code is used (§5.1.2) to resolve the class.
- If the first character of the fully qualified name of the constant pool entry to be resolved is a left bracket ("["), then the entry is a reference to an array class. Array classes are resolved specially (§5.1.3).

5.1.1 Current Class or Interface Not Loaded by a Class Loader

If a class or interface that has been loaded, and that was not loaded using a class loader, references a non-array class or interface C, then the following steps are performed to resolve the reference to C:

1. The class or interface C and its superclasses are first loaded (§2.16.2).
2. If class or interface C has not been loaded yet, the Java Virtual Machine will search for a file C.class and attempt to load class or interface C from that file. Note that there is no guarantee that the file C.class will actually contain the class or interface C, or that the file C.class is even a valid class file. It is also possible that class or interface C might have already been loaded, but not yet initialized. This phase of loading must detect the following errors:
   - If no file with the appropriate name can be found and read, class or interface resolution throws a NoClassDefFoundError.
   - Otherwise, if it is determined that the selected file is not a well-formed class file (pass 1 of §4.9.1), or is not a class file of a supported major or minor version (§4.1, class or interface resolution throws a NoClassDefFoundError.
   - Otherwise, if the selected class file did not actually contain the desired class or interface, class or interface resolution throws a NoClassDefFoundError.
   - Otherwise, if the selected class file does not specify a superclass and is not the class file for class Object, class or interface resolution throws a ClassFormatError.
3. If the superclass of the class being loaded has not yet been loaded, it is loaded using this step 1 recursively. Loading a superclass must detect any of the errors in step 1a, where this superclass is considered to be the class being loaded. Note that all interfaces must have java. lang. Object as their superclass, which must already have been loaded.
4. If loading class C and its superclasses was successful, the superclass (and thus its superclasses, if any) of class C is linked and initialized by applying steps -2–4 recursively.
5. The class C is linked (§2.16.3), that is, it is verified (§4.9) and prepared.
6. First, the class or interface C is verified to ensure that its binary representation is structurally valid (passes 2 and 3 of §4.9.1).[1] Verification may itself cause classes and interfaces to be loaded, but not initialized (to avoid circularity), using the procedure in step 1.

[1]Sun's JDK release 1.0.2 only verifies class files that have class loaders; it assumes that class files loaded locally are trusted and do not need verification.

   - If the class or interface C contained in class file C.class does not satisfy the static or structural constraints on valid class files listed in Section 4.8, "Constraints on Java Virtual Machine Code," class or interface resolution throws a VerifyError.
7. If the class file for class or interface C is successfully verified, the class or interface is prepared. Preparation involves creating the static fields for the class or interface and initializing those fields to their standard default values (§2.5.1). Preparation should not be confused with the execution of static initializers (§2.11); unlike execution of static initializers, preparation does not require the execution of any Java code. During preparation:
   - If a class that is not declared abstract has an abstract method, class resolution throws an AbstractMethodError.
8. Certain checks that are specific to individual Java Virtual Machine instructions, but that are logically related to this phase of constant pool resolution, are described in the documentation of those instructions. For instance, the getfield instruction resolves its field reference, and only afterward checks to see whether that field is an instance field (that is, it is not static). Such exceptions are still considered and documented to be linking, not runtime, exceptions.

9. Next, the class is initialized. Details of the initialization procedure are given in §2.16.5 and in *The Java Language Specification.*

If an initializer completes abruptly by throwing some exception E, and if the class of E is not Error or one of its subclasses, then a new instance of the class ExceptionInInitializerError, with E as the argument, is created and used in place of E.

If the Java Virtual Machine attempts to create a new instance of the class ExceptionInInitializerError but is unable to do so because an Out-Of--Memory-Error occurs, then the OutofMemoryError object is thrown instead.

10. Finally, access permissions to the class being resolved are checked:

If the current class or interface does not have permission to access the class or interface being resolved, class or interface resolution throws an IllegalAccess-Error. This condition can occur, for example, if a class that is originally declared public is changed to be private after another class that refers to the class has been compiled.

If none of the preceding errors were detected, constant pool resolution of the class or interface reference must have completed successfully. However, if an error was detected, one of the following must be true.

If some exception is thrown in steps 1–4, the class being resolved must have been marked as unusable or must have been discarded.

If an exception is thrown in step 5, the class being resolved is still valid and usable.

In either case, the resolution fails, and the class or interface attempting to perform the resolution is prohibited from accessing the referenced class or interface.

5.1.2 Current Class or Interface Loaded by a Class Loader

If a class or interface, loaded using a class loader, references a non-array class or interface C, then that same class loader is used to load C. The loadClass method of that class loader is invoked on the fully qualified path name (§2.7.9) of the class to be resolved. The value returned by the loadClass method is the resolved class. The remainder of the section describes this process in more detail.

Every class loader is an instance of a subclass of the abstract class ClassLoader. Applications implement subclasses of ClassLoader in order to extend the manner in which the Java Virtual Machine dynamically loads classes. Class loaders can be used to create classes that originate from sources other than files. For example, a class could be downloaded across a network, it could be generated on the fly, or it could be decrypted from a scrambled file.

The Java Virtual Machine invokes the loadClass method of a class loader in order to cause it to load (and optionally link and initialize) a class. The first argument to loadclass is the fully qualified name of the class to be loaded. The second argument is a boolean. The value false indicates that the specified class must be loaded, but not linked or initialized; the value true indicates the class must be loaded, linked, and initialized.

Implementations of class loaders are required to keep track of which classes they have already loaded, linked, and initialized:[2]

[2]Future implementations may change the API between the Java Virtual Machine and the class ClassLoader. Specifically, the Java Virtual Machine rather than the class loader will keep track of which classes and interfaces have been loaded by a particular class loader. One possibility is that the loadClass method will be called with a single argument indicating the class or interface to be loaded. The virtual machine will handle the details of linking and initialization and ensure that the class loader is not invoked with the same class or interface name multiple times.

If a class loader is asked to load (but not link or initialize) a class or interface that it has already loaded (and possibly already linked and initialized), then it should simply return that class or interface.

If a class loader is asked to load, link, and initialize a class or interface that it has already loaded but not yet linked and initialized, the class loader should not reload the class or interface, but should only link and initialize it.

If a class loader is asked to load, link, and initialize a class or interface that it has already loaded, linked, and initialized, the class loader should simply return that class or interface.

When the class loader's loadClass method is invoked with the name of a class or interface that it has not yet loaded, the class loader must perform one of the following two operations in order to load the class or interface:

The class loader can create an array of bytes representing the bytes of a file of class file format; it then must invoke the method defineClass of class ClassLoader on those bytes to convert them into a class or interface with this class loader as the class loader for the newly defined class. Invoking define-Class causes the Java Virtual Machine to perform step 1a of §5.1.1.

Invoking defineClass then causes the loadClass method of the class loader to be invoked recursively in order to load the superclass of the newly defined class or interface. The fully qualified path name of the superclass is derived from the super_class item in the class file format. When the superclass is loaded in, the second argument to loadClass is false, indicating that the superclass is not to be linked and initialized immediately.

The class loader can also invoke the static method findSystemClass in class ClassLoader with the fully qualified name of the class or interface to be loaded. Invoking this method causes the Java Virtual Machine to perform step 1 of §5.1.1. The resulting class file is not marked as having been loaded by a class loader.

After the class or interface and its superclasses have been loaded successfully, if the second argument to loadClass is true the class or interface is linked and initialized. This second argument is always true if the class loader is being called upon to resolve an entry in the constant pool of a class or interface. The class loader links and initializes a class or interface by invoking the method resolveClass in the class ClassLoader. Linking and initializing a class or interface created by a class loader is very similar to linking and initializing a class or interface without a class loader (steps 2–4 of 5.1.1):

First, the superclass of the class or interface is linked and initialized by calling the loadClass method of the class loader with the fully qualified name of the superclass as the first argument, and true as the second argument. Linking and initialization may result in the superclass's own superclass being linked and initialized. Linking and initialization of a superclass must detect any of the errors of step 3 of §5.1.1.

Next, the bytecode verifier is run on the class or interface being linked and initialized. The verifier may itself need classes or interfaces to be loaded, and if so, it loads them by invoking the loadclass method of the same class loader with the second argument being false. Since verification may itself cause classes or interfaces to be loaded (but not linked or initialized, to avoid circularity), it must detect the errors of step 1 of §5.1.1. for any classes or interfaces it attempts to load. Running the verifier may also cause the errors of step 3a of §5.1.1.

If the class file is successfully verified, the class or interface is then prepared (step 3b of §5.1.1) and initialized (step 4 of §5.1.1).

Finally, access permissions to the class or interface are checked (step 5 of §5.1.1). If the current class or interface does not have permission to access the class being resolved, class resolution throws an IllegalAccessError exception.

If none of the preceding errors were detected, loading, linking, and initialization of the class or interface must have completed successfully.

5.1.3 Array Classes

A constant pool entry tagged as CONSTANT_Class (§4.4.1) represents an array class if the first character of the UTF-8 string (§4.4.7) referenced by the name_index item of that constant pool entry is a left bracket ("["). The number of initial consecutive left brackets in the name represents the number of dimensions of the array class. Following the one or more initial consecutive left brackets is a field descriptor (§4.3.2) representing either a primitive type or a non-array reference type; this field descriptor represents the base type of the array class.

The following steps are performed to resolve an array class referenced from the constant pool of a class or interface:

1. Determine the number of dimensions of the array class and the field descriptor that represents the base type of the array class.
2. Determine the base type of the array class:
   If the field descriptor represents a primitive type (its first character is not "L"), that primitive type is the base type of the array class.
   If the field descriptor represents a non-array reference type (its first character is "L"), that reference type is the base type of the array class. The reference type is itself resolved using the procedures indicated above in §5.1.1 or in §5.1.2.
1. If an array class representing the same base type and the same number of dimensions has already been created, the result of the resolution is that array class. Otherwise, a new array class representing the indicated base type and number of dimensions is created.

5.2 Field and Method Resolution

A constant pool entry tagged as CONSTANT_Fieldref (§4.4.2) represents a class or instance variable (§2.9) or a (constant) field of an interface (§2.13.4). Note that interfaces do not have instance variables. A constant pool entry tagged as CONSTANT_Methodref (§4.4.2) represents a method of a class (a static method) or of a class instance (an instance method). References to interface methods are made using CONSTANT_InterfaceMethodref constant pool entries; resolution of such entries is described in §5.3.

To resolve a field reference or a method reference, the CONSTANT_Class (§4.4.1) entry representing the class of which the field or method is a member must first be successfully resolved (§5.1) Thus, any exception that can be thrown when resolving a CONSTANT_Class constant pool entry can also be thrown as a result of resolving a CONSTANT_Fieldref or CONSTANT_Methodref entry. If the CONSTANT_Class entry representing the class or interface can be successfully resolved, exceptions relating to the linking of the method or field itself can be thrown. When resolving a field reference:

If the referenced field does not exist in the specified class or interface, field resolution throws a NoSuchFieldError.

Otherwise, if the current class does not have permission to access the referenced field, field resolution throws an IllegalAccessError exception.

If resolving a method:

If the referenced method does not exist in the specified class or interface, field resolution throws a NoSuchMethodError.

Otherwise, if the current class does not have permission to access the method being resolved, method resolution throws an IllegalAccessError exception.

5.3 Interface Method Resolution

A constant pool entry tagged as CONSTANT_InterfaceMethodref (§4.4.2) represents a call to an instance method declared by an interface. Such a constant pool entry is resolved by converting it into a machine-dependent internal format. No error or exception is possible except for those documented in §6.3.

5.4 String Resolution

A constant pool entry tagged as CONSTANT_String (§4.4.3) represents an instance of a string literal (§2.3), that is, a literal of the built-in type java.lang.String. The Unicode characters (§2.1) of the string literal represented by the CONSTANT_String entry are found in the CONSTANT_Utf8 (§4.4.7) constant pool entry that the CONSTANT_String entry references.

The Java language requires that identical string literals (that is, literals that contain the same sequence of Unicode characters) must reference the same instance of class String. In addition, if the method intern is called on any string, the result is a reference to the same class instance that would be returned if that string appeared as a literal. Thus,

("a" + "b" + "c").intern( ) == "abc"

must have the value true.[3]

[3]String literal resolution is not implemented correctly in Sun's JDK release 1.0.2. In that implementation of the Java Virtual Machine, resolving a CONSTANT_String in the constant pool always allocates a new string. Two string literals in two different classes, even if they contained the identical sequence of characters, would never be == to each other. A string literal could never be == to a result of the intern method.

To resolve a constant pool entry tagged CONSTANT_String, the Java Virtual Machine examines the series of Unicode characters represented by the UTF-8 string that the CONSTANT_String entry references.

If another constant pool entry tagged CONSTANT_String and representing the identical sequence of Unicode characters has already been resolved, then the result of resolution is a reference to the instance of class String created for that earlier constant pool entry.

Otherwise, if the method intern has previously been called on an instance of class String containing a sequence of Unicode characters identical to that represented by the constant pool entry, then the result of resolution is a reference to that same instance of class String.

Otherwise, a new instance of class String is created containing the sequence of Unicode characters represented by the CONSTANT_String entry; that class instance is the result of resolution.

No error or exception is possible during string resolution except for those documented in §6.3.

5.5 Resolution of Other Constant Pool Items

Constant pool entries that are tagged CONSTANT_Integer or CONSTANT_Float (§4.4.4), CONSTANT_Long or CONSTANT_Double (§4.4.5) all have values that are directly represented within the constant pool. Their resolution cannot throw exceptions except for those documented in §6.3.

Constant pool entries that are tagged CONSTANT_NameAndType (§4.4.6), and CONSTANT_Utf8 (§4.4.7) are never resolved directly. They are only referenced directly or indirectly by other constant pool entries.

We claim:

1. A method of pre-processing class files comprising:

determining plurality of duplicated elements in a plurality of class files;

forming a shared table comprising said plurality of duplicated elements;

removing said duplicated elements from said plurality of class files to obtain a plurality of reduced class files; and

forming a multi-class file comprising said plurality of reduced class files and said shared table.

2. The method of claim 1, further comprising:

computing an individual memory allocation requirement for each of said plurality of reduced class files;

computing a total memory allocation requirement for said plurality of class files from said individual memory allocation requirement of each of said plurality of reduced class files; and

storing said total memory allocation requirement in said multi-class file.

3. The method of claim 2, further comprising:

reading said total memory allocation requirement from said multi-class file;

allocating a portion of memory based on said total memory allocation requirement; and

loading said reduced class files and said shared table into said portion of memory.

4. The method of claim 3, further comprising:

accessing said shared table in said portion of memory to obtain one or more elements not found in one or more of said reduced class files.

5. The method of claim 1, wherein said step of determining a plurality of duplicated elements comprises:

determining one or more constants shared between two or more class files.

6. The method of claim 5, wherein said step of forming a shared table comprises:

forming a shared constant table comprising said one or more constants shared between said two or more class files.

7. A computer program product comprising:

a computer usable medium having computer readable program code embodied therein for pre-processing class files, said computer program product comprising:

computer readable program code configured to cause a computer to determine a plurality of duplicated elements in a plurality of class files;

computer readable program code configured to cause a computer to form a shared table comprising said plurality of duplicated elements;

computer readable program code configured to cause a computer to remove said duplicated elements from said plurality of class files to obtain a plurality of reduced class files; and

computer readable program code configured to cause a computer to form a multi-class file comprising said plurality of reduced class files and said shared table.

8. The computer program product of claim 7, further comprising:

computer readable program code configured to cause a computer to compute an individual memory allocation requirement of each of said plurality of reduced class files;

computer readable program code configured to cause a computer to compute a total memory allocation requirement of said plurality of class files from said individual memory allocation requirement of each of said plurality of reduced class files; and

computer readable program code configured to cause a computer to store said total memory allocation requirement in said multi-class file.

9. The computer program product of claim 8, further comprising:

computer readable program code configured to cause a computer to read said total memory allocation requirement from said multi-class file;

computer readable program code configured to cause a computer to allocate a portion of memory based on said total memory allocation requirement; and

computer readable program code configured to cause a computer to load said reduced class files and said shared table into said portion of memory.

10. The computer program product of claim 9, further comprising:

computer readable program code configured to cause a computer to access said shared table in said portion of memory to obtain one or more elements not found in one or more of said reduced class files.

11. The computer program product of claim 7, wherein said computer readable program code configured to cause a computer to determine said plurality of duplicated elements comprises:

computer readable program code configured to cause a computer to determine one or more constants shared between two or more class files.

12. The computer program product of claim 11, wherein said computer readable program code configured to cause a computer to form said shared table comprises:

computer readable program code configured to cause a computer to form a shared constant table comprising said one or more constants shared between said two or more class files.

13. An apparatus comprising:

a processor;

a memory coupled to said processor;

a plurality of class files stored in said memory;

a process executing on said processor, said process configured to form a multi-class file comprising:

a plurality of reduced class files obtained from said plurality of class files by removing one or more elements that are duplicated between two or more of said plurality of class files; and

a shared table comprising said duplicated elements.

14. The apparatus of claim 13, wherein said multi-class file further comprises a memory requirement, said memory requirement being computed by said process.

15. The apparatus of claim 13, wherein said duplicated elements comprise elements of constant pools of respective class files, said shared table comprising a shared constant pool.

16. The apparatus of claim 13, further comprising:

a virtual machine having a class loader and a runtime data area, said class loader configured to obtain and load said multi-class file into said runtime data area.

17. The apparatus of claim 16, wherein said class loader is configured to allocate a portion of said runtime data area based on said memory requirement in said multi-class file.

18. The apparatus of claim 17, wherein said class loader is configured to load said plurality of reduced class files and said shared table into said portion of said runtime data area.

19. The apparatus of claim 16, wherein said virtual machine is configured to access said shared table when a desired element associated with a first class file is not present in a corresponding one of said plurality of reduced class files.

20. A memory configured to store data for access by a virtual machine executing in a computer system, comprising:

a data structure stored in said memory, said data structure comprising:

a plurality of reduced class files associated with a plurality of corresponding classes, said plurality of reduced class files configured to be loaded by the virtual machine for execution of said plurality of classes;

a shared table comprising one or more elements that are duplicated between two or more of said plurality of classes, said shared table configured to be loaded into the virtual machine to be accessed for said duplicated elements; and

a memory requirement value configured to be read by a class loader of the virtual machine to allocate a portion of a runtime data area for loading said plurality of reduced class files and said shared table.

21. The memory of claim 20, wherein said duplicated elements are removed from said plurality of reduced class files.

22. The memory of claim 20, wherein said duplicated elements comprise constants and said shared table comprises a shared constant pool.

23. The memory of claim 20, wherein said memory requirement value is computed from individual memory requirements of said plurality of reduced class files and a memory requirement of said shared table.

* * * * *

EXHIBIT D

US007426720B1

(12) **United States Patent**
Fresko

(10) **Patent No.: US 7,426,720 B1**
(45) **Date of Patent: *Sep. 16, 2008**

(54) **SYSTEM AND METHOD FOR DYNAMIC PRELOADING OF CLASSES THROUGH MEMORY SPACE CLONING OF A MASTER RUNTIME SYSTEM PROCESS**

(75) Inventor: **Nedim Fresko**, San Francisco, CA (US)

(73) Assignee: **Sun Microsystems, Inc.**, Santa Clara, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 745 days.

This patent is subject to a terminal disclaimer.

(21) Appl. No.: **10/745,023**

(22) Filed: **Dec. 22, 2003**

(51) **Int. Cl.**
G06F 9/45 (2006.01)

(52) **U.S. Cl.** **717/140**; 717/151; 717/152; 717/153; 718/1

(58) **Field of Classification Search** 717/151–153, 717/140; 718/1
See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

6,374,286 B1 * 4/2002 Gee et al. 718/108
6,823,509 B2 * 11/2004 Webb 718/1
6,829,761 B1 * 12/2004 Sexton et al. 717/165
2003/0088604 A1 * 5/2003 Kuck et al. 709/1

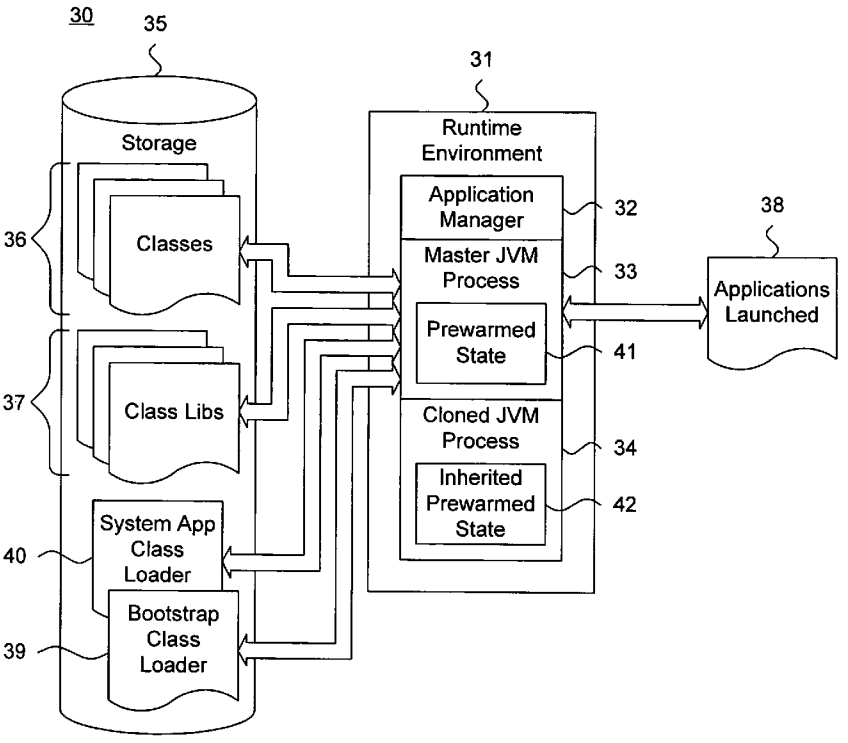
* cited by examiner

Primary Examiner—Wei Y. Zhen
Assistant Examiner—Junchun Wu
(74) *Attorney, Agent, or Firm*—Park, Vaughan & Fleming LLP

(57) **ABSTRACT**

A system and method for dynamic preloading of classes through memory space cloning of a master runtime system process is presented. A master runtime system process is executed. A representation of at least one class is obtained from a source definition provided as object-oriented program code. The representation is interpreted and instantiated as a class definition in a memory space of the master runtime system process. The memory space is cloned as a child runtime system process responsive to a process request and the child runtime system process is executed, inheriting the memory state of the parent, which reflects the data structures and state corresponding to the preloaded classes.

22 Claims, 11 Drawing Sheets



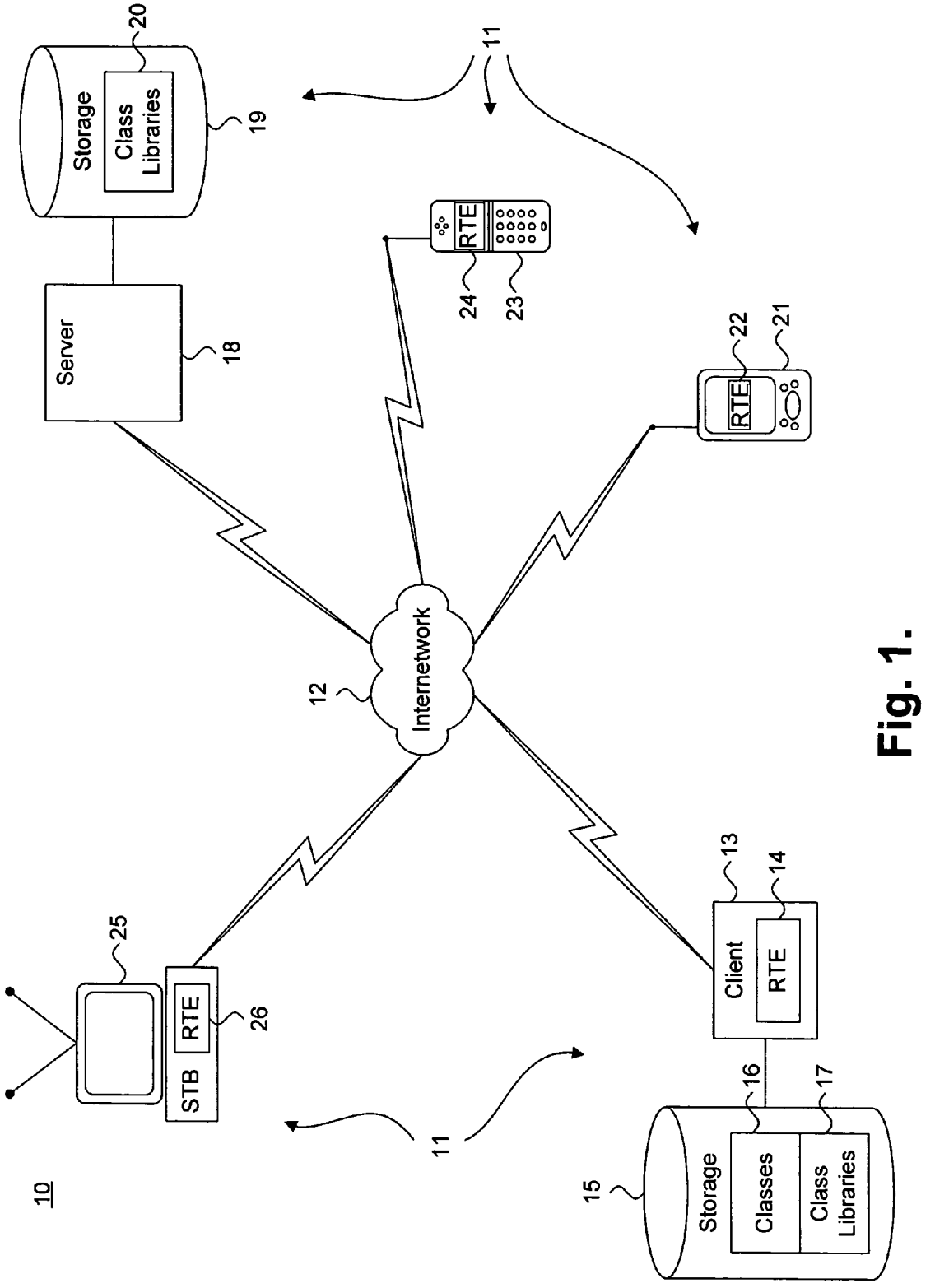

10
Storage
Class Libraries
20
19
11
Server
18
RTE
24
23
22
21
RTE
Internetwork
12
25
STB
RTE
26
Client
RTE
13
14
16
17
15
Storage
Classes
Class Libraries
11

Fig. 1.

Fig. 2.

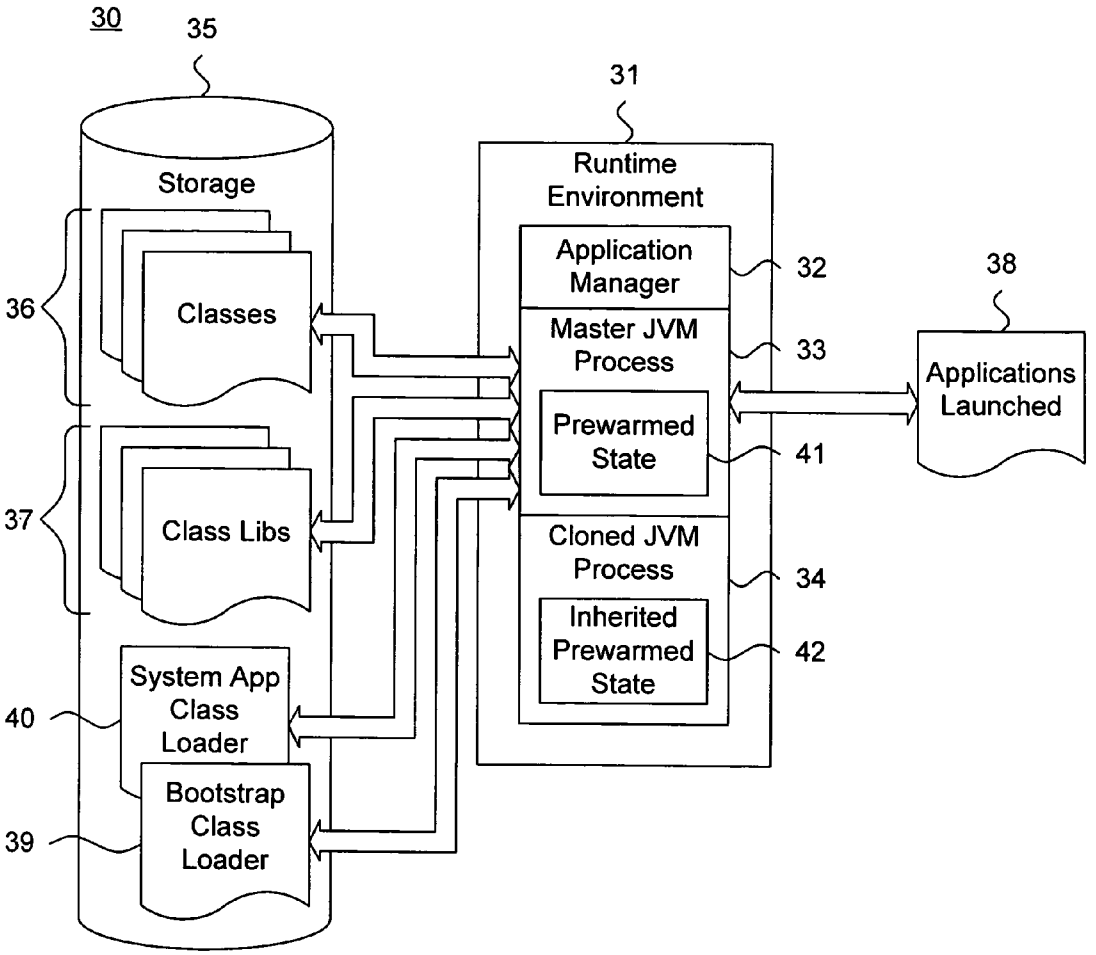

30
35
31
Storage
Runtime Environment
Application Manager
32
38
36
Classes
Master JVM Process
33
Applications Launched
Prewarmed State
41
37
Class Libs
Cloned JVM Process
34
Inherited Prewarmed State
42
System App Class Loader
40
Bootstrap Class Loader
39

Fig. 3.

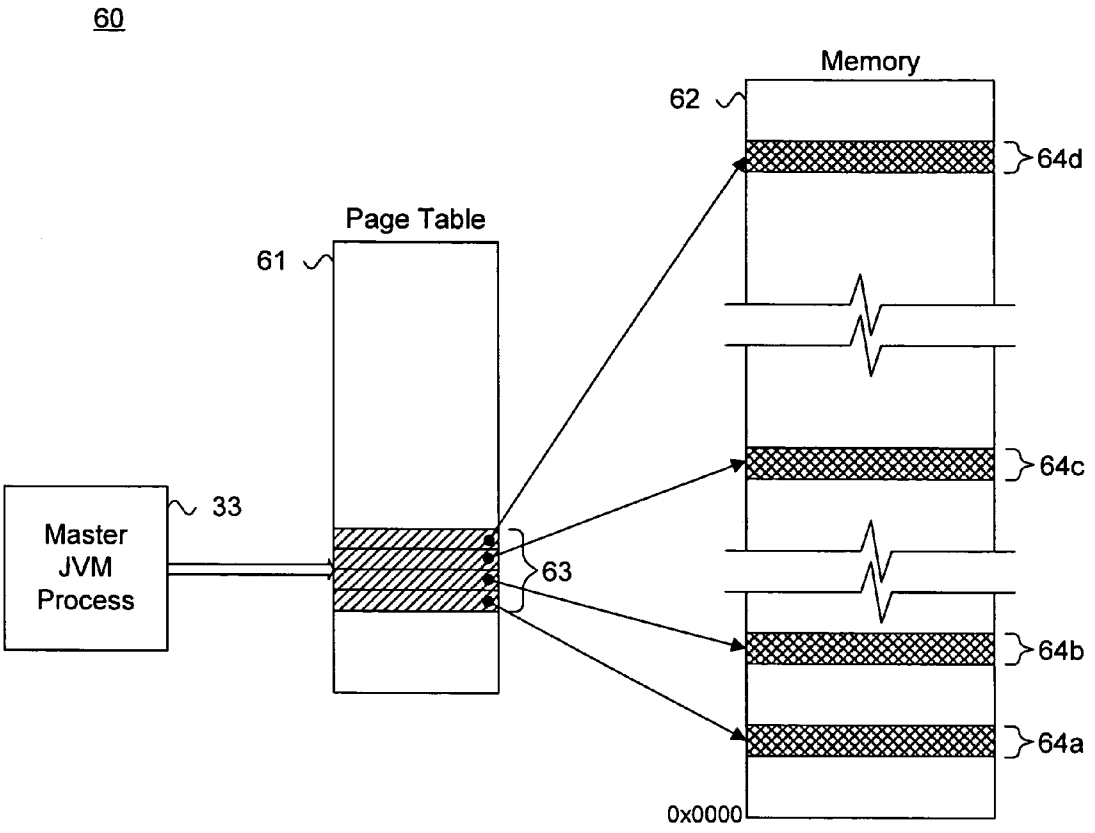

60
Memory
62
64d
Page Table
61
64c
33
Master
JVM
Process
63
64b
64a
0x0000

Fig. 4.

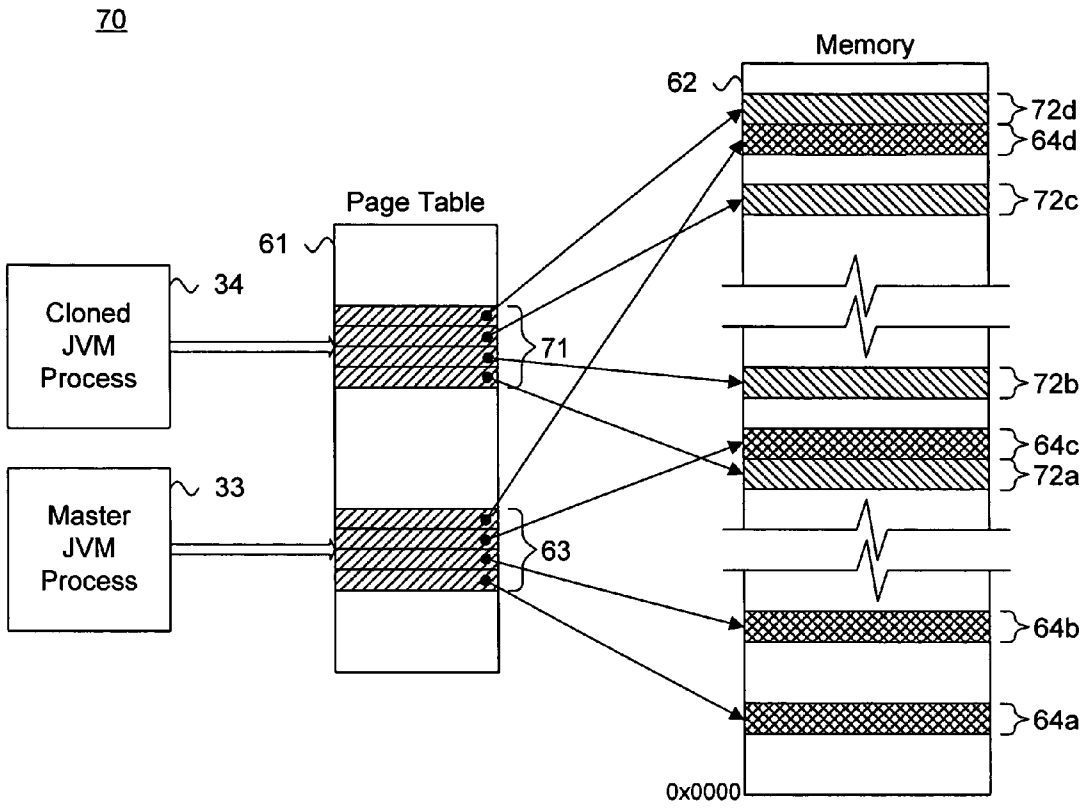

70
Memory
62
72d
64d
72c
Page Table
61
34
Cloned JVM Process
71
72b
64c
72a
33
Master JVM Process
63
64b
64a
0x0000

Fig. 5A.

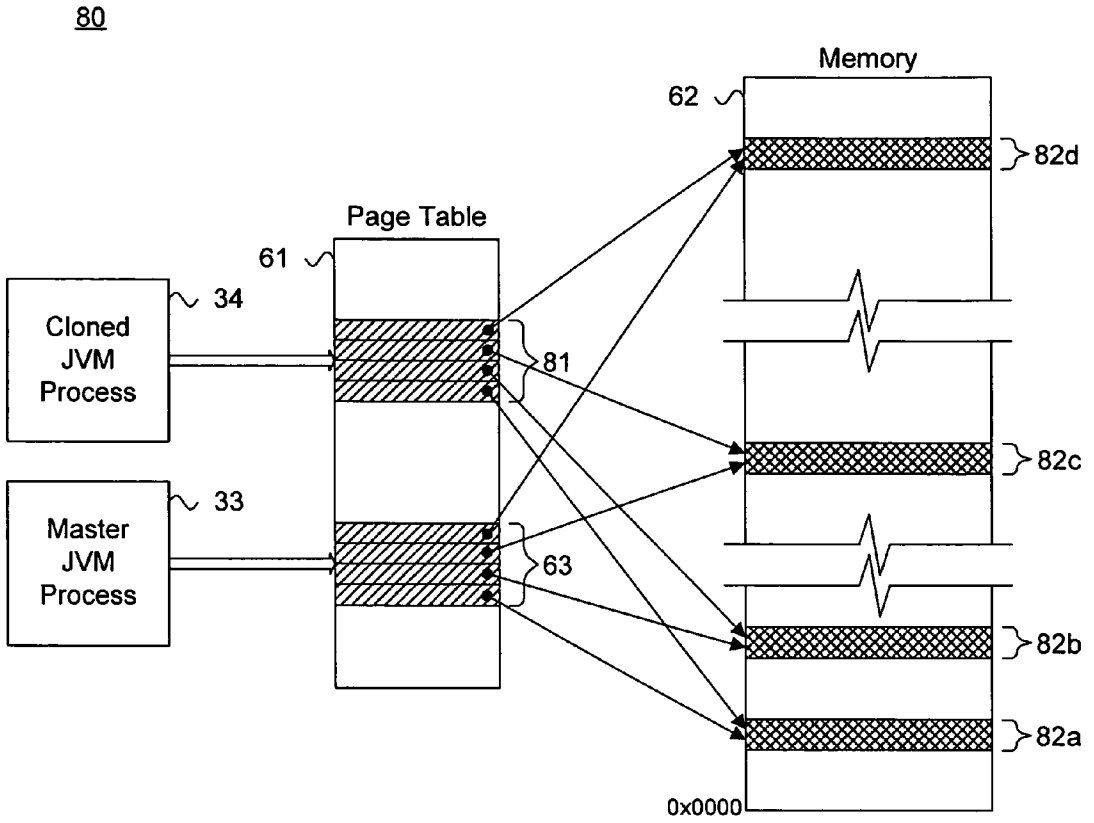

80
Memory
62
82d
Page Table
61
34
Cloned JVM Process
81
82c
33
Master JVM Process
63
82b
82a
0x0000

Fig. 5B.

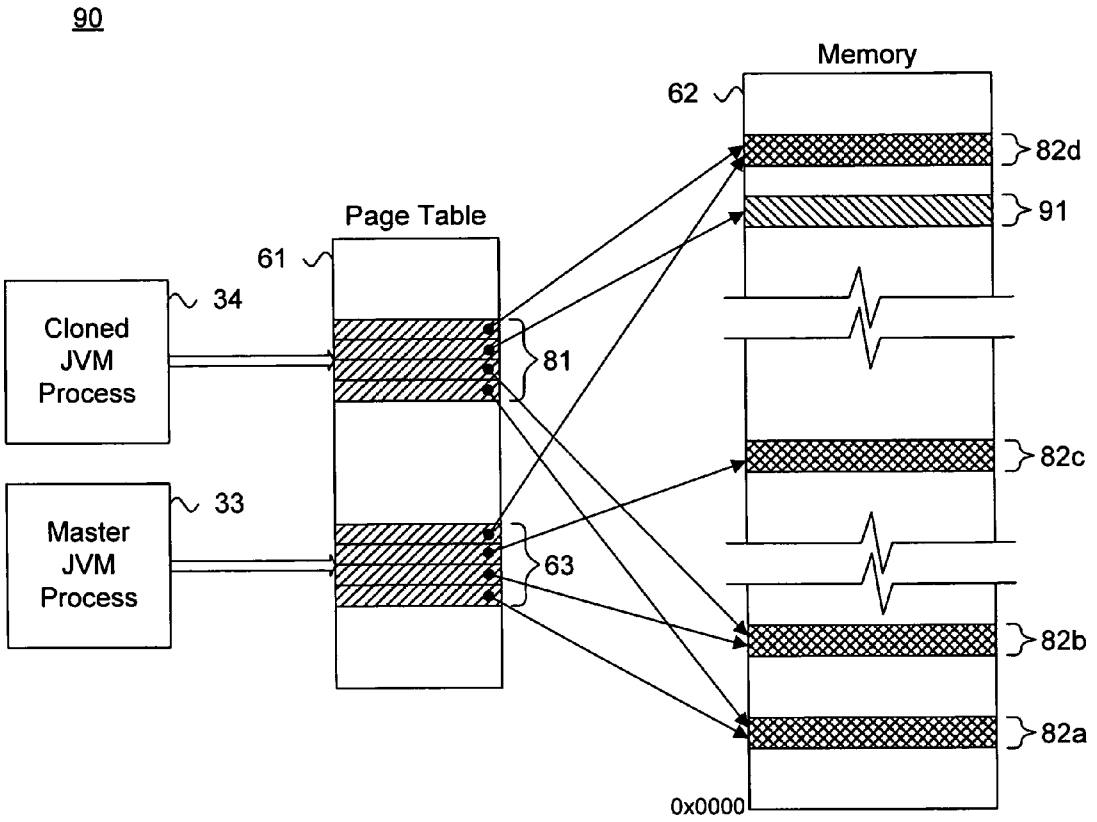

90
Memory
62
82d
91
Page Table
61
34
Cloned JVM Process
81
82c
33
Master JVM Process
63
82b
82a
0x0000

Fig. 6.

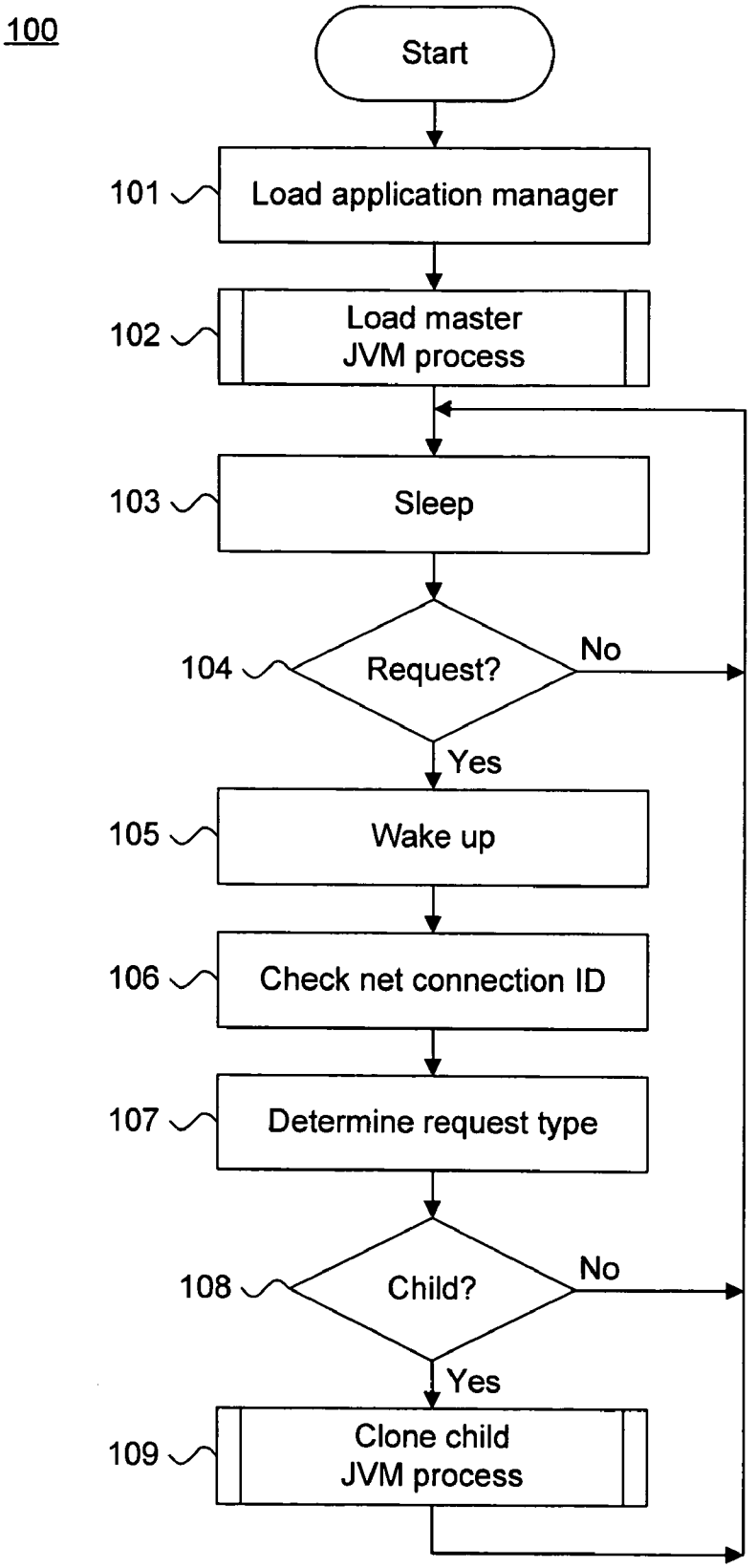

100
Start
101
Load application manager
102
Load master JVM process
103
Sleep
104
Request?
No
Yes
105
Wake up
106
Check net connection ID
107
Determine request type
108
Child?
No
Yes
109
Clone child JVM process

Fig. 7.

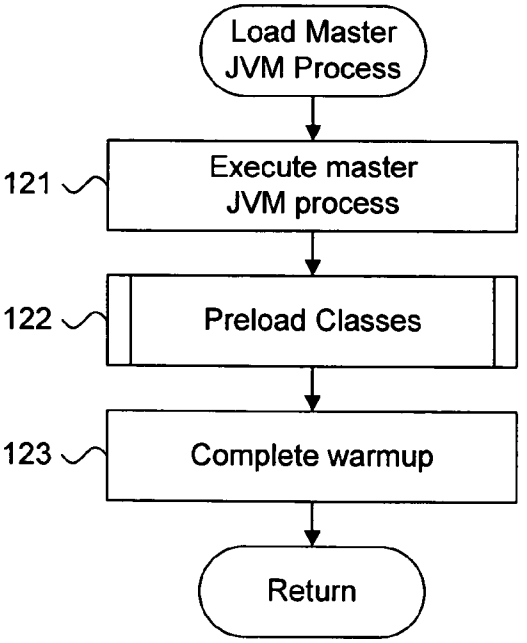

120
Load Master JVM Process
121
Execute master JVM process
122
Preload Classes
123
Complete warmup
Return

Fig. 8.

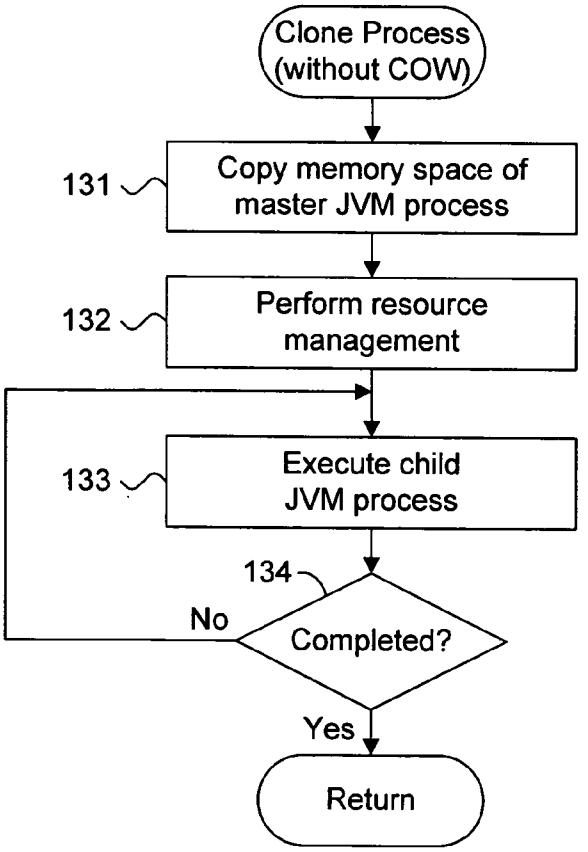

130
Clone Process (without COW)
131
Copy memory space of master JVM process
132
Perform resource management
133
Execute child JVM process
134
No
Completed?
Yes
Return

Fig. 9.

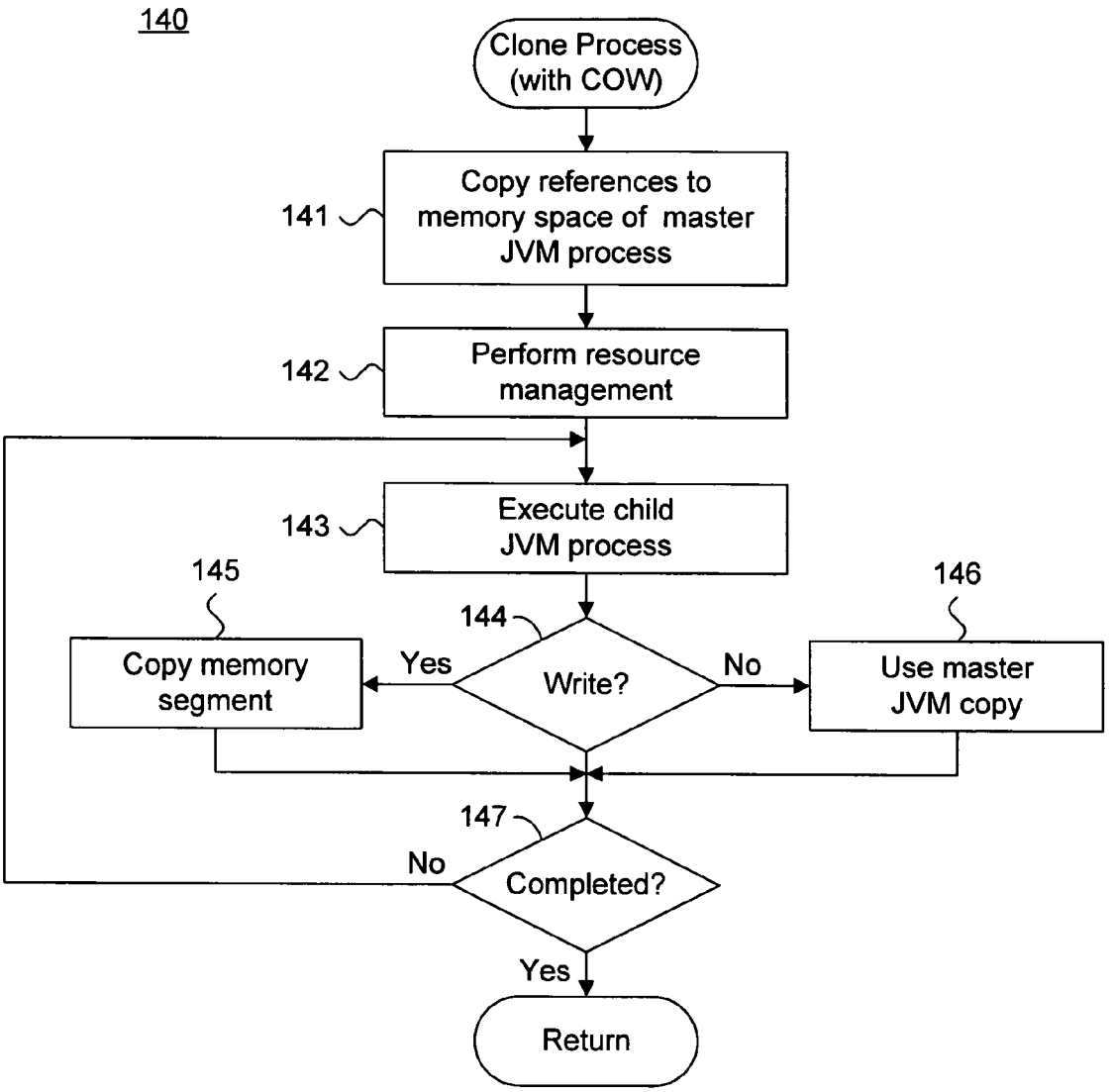

140
Clone Process (with COW)
141
Copy references to memory space of master JVM process
142
Perform resource management
143
Execute child JVM process
144
Write?
Yes
No
145
Copy memory segment
146
Use master JVM copy
147
Completed?
No
Yes
Return

Fig. 10.

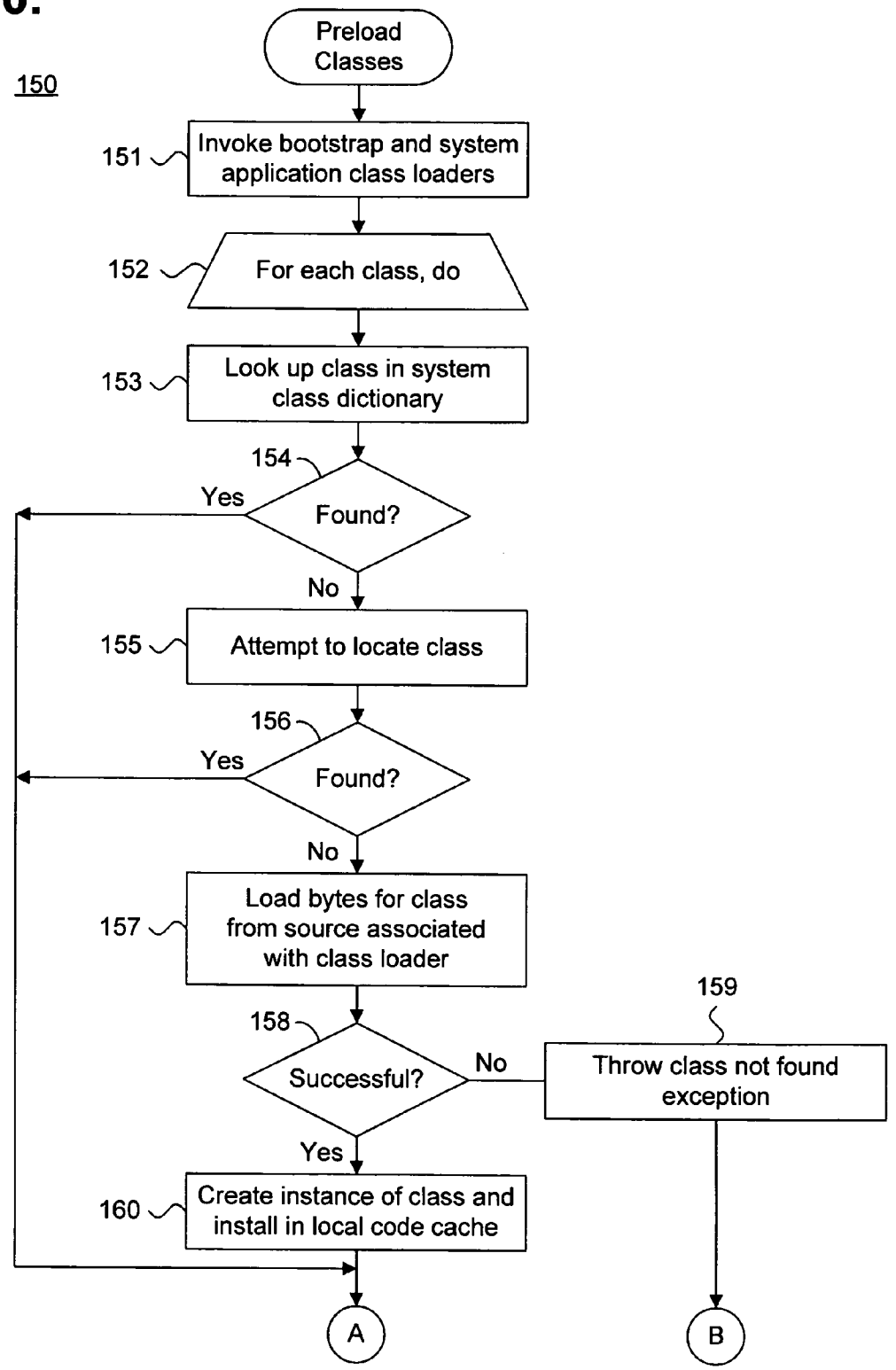
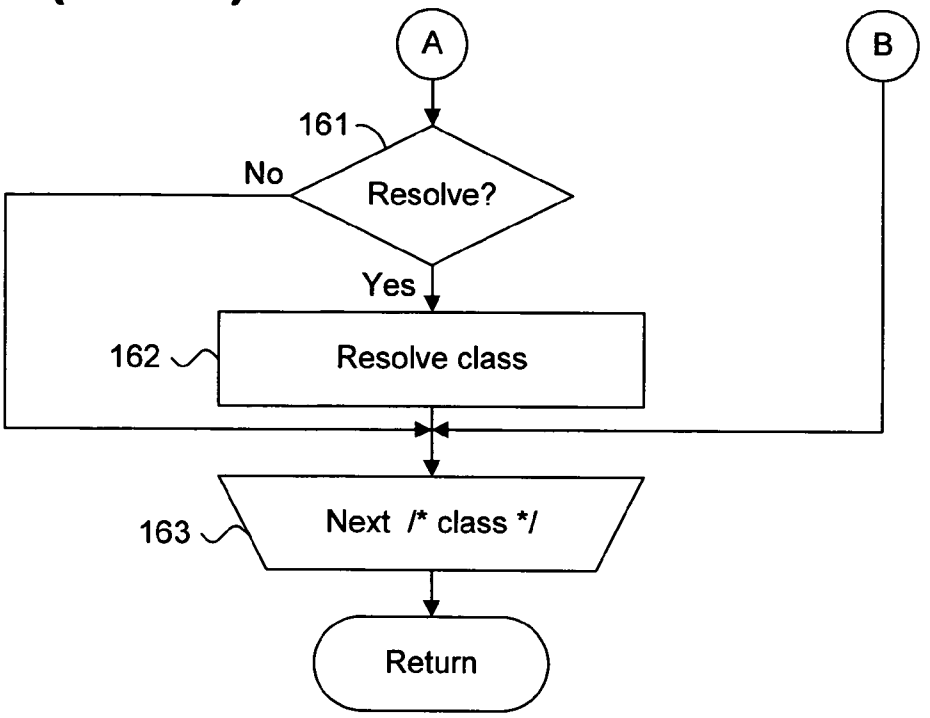

150
Preload Classes
151
Invoke bootstrap and system application class loaders
152
For each class, do
153
Look up class in system class dictionary
154
Found?
Yes
No
155
Attempt to locate class
156
Found?
Yes
No
157
Load bytes for class from source associated with class loader
158
Successful?
No
159
Throw class not found exception
Yes
160
Create instance of class and install in local code cache
A
B

Fig. 10 (Cont.).


A
B
161
No
Resolve?
Yes
162
Resolve class
163
Next /* class */
Return

SYSTEM AND METHOD FOR DYNAMIC PRELOADING OF CLASSES THROUGH MEMORY SPACE CLONING OF A MASTER RUNTIME SYSTEM PROCESS

FIELD OF THE INVENTION

The invention relates in general to class preloading and, in particular, to a system and method for dynamic preloading of classes through memory space cloning of a master runtime system process.

BACKGROUND OF THE INVENTION

Recent advances in microprocessor design and component integration have enabled a wide range of devices to offer increasingly complex functionality and "soft" features. Soft features include software applications that enhance and customize the operation of a device. These devices include standard computing devices, such as desktop and laptop computers, portable computing devices, such as personal data assistants, and consumer devices, such as cellular telephones, messaging pagers, gaming consoles, and set top boxes. Most devices now include an operating system to support the soft features and other extensions.

The increased capabilities offered by these software-upgradeable devices have also created certain user expectations. Often, users are not technically savvy and are intolerant of performance compromises occasioned by architectural challenges, such as slow or inconsistent application performance. Similarly, users generally expect to be able to access a host of separate applications, which are implemented at the system level through multitasking. For users, widely available software applications assure a positive experience through consistency and increased exposure across multiple platforms. However, for software developers, engineering software applications for disparate computing platforms entails increased development costs and on-going support and upgrade commitments for each supported architecture.

Managed code platforms provide one solution to software developers seeking to support multiple platforms by presenting a machine-independent and architecture-neutral operating environment. Managed code platforms include programming language compilers and interpreters executed by an operating system as user applications, but which provide virtual runtime environments within which compatible applications can operate. For instance, applications written in the Java programming language, when combined with a Java virtual machine (JVM) runtime environment, can operate on heterogeneous computer systems independent of machine-specific environment and configuration settings. An overview of the Java programming language is described in P. van der Linden, "Just Java," Ch. 1, Sun Microsystems, Inc. (2d ed. 1997), the disclosure of which is incorporated by reference. JVMs are a critical component to the overall Java operating environment, which can be ported to the full range of computational devices, including memory-constrained consumer devices.

Managed code platforms are generally designed for the monotonic execution of a single application instance. Multiple instances of a managed code platform are executed to simulate multitasking behavior. Such forced concurrency, however, creates several performance problems. First, each instance incurs a startup transient. Executable and startup data must be read from slow persistent storage, which results in slow initial application performance. Similarly, memory is not shared between instances and each additional instance

increases the overall memory footprint of the platform by separately loading and instantiating classes, generally problematic in memory-constrained systems. Moreover, data dependencies and deferred initialization of system state can result in non-deterministic execution patterns. Finally, each instance independently determines the relative importance of executing methods and compiles machine code on an ad hoc basis, often causing inconsistent application performance.

Deferred class loading is sometimes necessitated by the dynamic nature of the object oriented languages involved. Dynamic class loading can also adversely affect performance and cause nondeterministic execution behavior. To help improve runtime performance, managed code platforms lazily defer class loading until a class is actually referenced. Deferred class loading conserves the time required to load a class by delaying class loading and compilation until, and if, the class is actually needed. Deferred class loading sacrifices runtime performance for improved application startup. However, for near real time applications, deferred class loading causes non-deterministic execution behavior that increases worst case performance by the longest class loading execution thread. Similarly, deferred class loading exacerbates the resource usage of multiple application instances that each requires the same classes by duplicatively performing identical operations and needlessly consuming memory that could be conserved, if the memory state were shared.

Static preloading of classes and interfaces is currently supported in many Java virtual machines, which allows a build-time tool to pre-process and preload classes and to link the classes into the JVM static executable image before JVM startup. However, static preloading can result in large executable sizes and can be problematic for resource constrained devices, where boot startup time is critical and a combination of slower processor and persistent storage and modest memory can cause significant boot times.

Therefore, there is a need for an approach to providing class preloading in a managed code platform, such as the Java operating environment, to provide concurrently executable applications that share warmed up memory state and to minimize worst case performance.

SUMMARY OF THE INVENTION

A managed code platform is executed in an application framework that supports the spawning of multiple and independent isolated user applications. Preferably, the application framework supports the cloning of the memory space of each user application using copy-on-write semantics. The managed code platform includes a master runtime system process, such as a virtual machine, to interpret machine-portable code defining compatible applications. An application manager also executes within the application framework and is communicatively interfaced to the master runtime system process through an inter-process communication mechanism. The application framework logically copies the master runtime system process context upon request by the application framework to create a child runtime system process through process cloning. The context of the master runtime system process stored in memory is inherited by the child runtime system process as prewarmed state and cached code. When implemented with copy-on-write semantics, the process cloning creates a logical copy of references to the master runtime system process context. Segments of the referenced master runtime system process context are lazily copied only upon an attempt by the child runtime system process to modify the referenced context. During initialization, the master runtime system process preloads classes and interfaces

likely to be required by user applications at runtime. The classes and interfaces are identified through profiling by ranking a set of classes according to a predetermined criteria, such as described in commonly-assigned U.S. patent application Ser. No. 09/970,661, filed Oct. 5, 2001, pending, the disclosure of which is incorporated by reference. An example of a suitable managed code platform and runtime system process are the Java operating environment and Java virtual machine (JVM) architecture, as licensed by Sun Microsystems, Inc., Palo Alto, Calif.

One embodiment provides a system and method for dynamic preloading of classes through memory space cloning of a master runtime system process. A master runtime system process is executed. A representation of at least one class is obtained from a source definition provided as object-oriented program code. The representation is interpreted and instantiated as a class definition in a memory space of the master runtime system process. The memory space is cloned as a child runtime system process responsive to a process request and the child runtime system process is executed.

The use of the process cloning mechanism provided by the underlying application framework provides several benefits in addition to resolving the need for efficient concurrent application execution of machine portable code. The inheritance of prewarmed state through the cloning of the master runtime process context provides inter-process sharing of preloaded classes. Similarly, each child runtime system process executes in isolation of each other process, thereby providing strong resource control through the system level services of the application framework. Isolation, reliable process invocation and termination, and resource reclamation are available and cleanly provided at an operating system level. In addition, process cloning provides fast user application initialization and deterministic runtime behavior, particularly for environments providing process cloning with copy-on-write semantics. Finally, for non-shareable segments of the master runtime system process context, actual copying is deferred until required through copy-on-write semantics, which avoids impacting application performance until, and if, the segment is required.

Still other embodiments of the invention will become readily apparent to those skilled in the art from the following detailed description, wherein are described embodiments of the invention by way of illustrating the best mode contemplated for carrying out the invention. As will be realized, the invention is capable of other and different embodiments and its several details are capable of modifications in various obvious respects, all without departing from the spirit and the scope of the invention. Accordingly, the drawings and detailed description are to be regarded as illustrative in nature and not as restrictive.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. **1** is a functional block diagram showing, by way of example, runtime environments implemented on a plurality of heterogeneous devices.

FIG. **2** is a block diagram showing a system for dynamic preloading of classes through memory space cloning of a master runtime system process, in accordance with the invention.

FIG. **3** is a block diagram showing, by way of example, a master JVM process mapped into memory.

FIG. **4** is a block diagram showing, by way of example, a master JVM process and a cloned JVM process mapped into memory through memory space cloning.

FIGS. **5**A-B are block diagrams showing, by way of example, a master JVM process and a cloned JVM process mapped into memory through memory space cloning with copy-on-write semantics.

FIG. **6** is a flow diagram showing a method for dynamic preloading of classes through memory space cloning of a master runtime system process, in accordance with the invention.

FIG. **7** is a flow diagram showing the routine for loading a master JVM process for use in the method of FIG. **6**.

FIG. **8** is a flow diagram showing the routine for cloning a process without copy-on-write semantics for use in the method of FIG. **6**.

FIG. **9** is a flow diagram showing the routine for cloning a process with copy-on-write semantics for use in the method of FIG. **6**.

FIG. **10** is a flow diagram showing the routine for preloading a class for use in the routine of FIG. **7**.

DETAILED DESCRIPTION

System Overview

FIG. **1** is a functional block diagram **10** showing, by way of example, runtime environments (RTEs) **14**, **22**, **24**, **26** implemented on a plurality of heterogeneous devices **11**. Each heterogeneous device **11** provides a managed code platform, such as the Java operating environment, executing in a runtime environment **14**, **22**, **24**, **26**, as further described below with reference to FIG. **2**. The heterogeneous devices **11** include, nonexclusively, a client computer system **13**, such as a desktop or laptop computer system. Each client **13** is operatively coupled to a storage device **15** and maintains a set of classes **16** and class libraries **17**, which respectively define code modules that specify data structures and sets of methods that operate on the data, and shareable collections of the modules. The heterogeneous devices **11** also include portable computing devices, including personal data assistants **21**, and consumer devices, such as cellular telephones **23** and set top boxes (STB) **25**. Finally, a server **18** is operatively coupled to a storage device **19** in which globally shareable class libraries **20** are maintained. Each of the heterogeneous devices **11** can interface via a network **12**, which includes conventional hardwired and wireless network configurations. Other types of heterogeneous devices **11** and various network configurations, arrangements, and topologies are possible.

Each heterogeneous device **11** includes an operating system to manage resources, provide access to peripheral devices, allocate memory resources, and control program execution and termination. Each operating system supports a process cloning mechanism that spawns multiple and independent isolated user applications by cloning the memory space of specifiable processes. An example of a process cloning mechanism suitable for use in the present invention is the fork( ) system call provided by the Unix or Linux operating systems, such as described in M. J. Bach, "The Design Of The Unix Operating System," Ch. **7**, Bell Tele. Labs., Inc. (1986), the disclosure of which is incorporated by reference. The process invoking the fork( ) system call is known as the parent process and the newly created process is called the child process. The operating system assigns a separate process identifier to the child process, which executes as a separate process. The operating system also creates a logical copy of the context of the parent process by copying the memory space of the parent process into the memory space of the child process. In a copy-on-write variant of the fork( ) system call, the operating system only copies references to the memory

space and defers actually copying individual memory space segments until, and if, the child process attempts to modify the referenced data of the parent process context. The copy-on-write fork( ) system call is faster than the non-copy-on-write fork( ) system call and implicitly shares any data not written into between the parent and child processes.

System for Preloading Classes

FIG. **2** is a block diagram **30** showing a system for dynamic preloading of classes through memory space cloning of a master runtime system process **33**, in accordance with the invention. Although described with specific reference to classes, other forms of structured static data could also be preloaded, including data structures, processes, functions, subroutines, interfaces, and the like. The system consists of a runtime environment **31** and individual classes **36** and class libraries **37** that form the overall core managed code platform. By way of example, the system is described with reference to the Java operating environment, although other forms of managed code platforms that execute applications preferably written in an object oriented programming language, such as the Java programming language, could also be used.

The exemplary runtime environment **31** includes an application manager **32**, master Java virtual machine (JVM) process **33** and zero or more cloned JVM processes **34**. The master JVM process **33** and cloned JVM processes **34** respectively correspond to a master runtime system process and child runtime system processes. The master runtime system process, preferably provided as a virtual machine, interprets machine-portable code defining compatible applications. The runtime environment **31** need not execute cloned JVM processes **34**, which are only invoked upon request by the application manager **32**.

The runtime environment **31** executes an application framework that spawns multiple independent and isolated user application process instances by preferably cloning the memory space of a master runtime system process. The example of an application framework suitable for use in the present invention is the Unix operating system, such as described generally in M. J. Bach, supra at Ch. 2, the disclosure of which is incorporated by reference.

The application manager **32** presents a user interface through which individual applications can be selected and executed. The application manager **32** and master JVM process **33** preferably communicate via an inter-process communication (IPC) mechanism, such as a pipe or a socket. The master JVM process **33** is started at device boot time.

Upon initialization, the master JVM process **33** reads an executable process image from the storage device **35** and performs bootstrapping operations. These operations include preloading the classes **36** and classes defined in the class libraries **37**, as further described below with reference to FIG. **10**. Thus, upon completion of initialization, the memory image of the master JVM process **33** resembles that of an initialized, primed and warmed up JVM process with key classes stored in the master JVM process context as prewarmed state **41**. Preferably, the prewarmed state **41** is stored as read only data.

Following the initialization, the master JVM process **33** idles, that is, "sleeps" in an inactive state, while awaiting further instructions from the application manager **32**. The master JVM process **33** awakens in response to requests received from the application manager **32** to execute applications. The application manager **32** sends a request to the master JVM process **33**, including standard command line parameters, such as application name, class path, and application arguments. The master JVM process **33** awakens and creates a cloned JVM process **34** as a new cloned process instance of the master JVM process **33** using the process cloning mechanism of the underlying operating system. The context of the master JVM process **33** stored in memory as prewarmed state **41** is inherited by the cloned JVM process **34** as inherited prewarmed state **42**, thereby saving initialization and runtime execution times and providing deterministic execution behavior. Following the "cloning" of the cloned JVM process **34**, the master JVM process **33** records the launched application in an applications launched list **38** and returns to an inactive sleep state.

When implemented with copy-on-write semantics, the process cloning creates a logical copy of only the references to the master JVM process context. Segments of the referenced master JVM process context are lazily copied only upon an attempt by the cloned JVM process to modify the referenced context. Therefore, as long as the cloned JVM process does not write into a memory segment, the segment remains shared between parent and child processes.

The master JVM process **33** recognizes the following basic commands received from the application manager **32** through the IPC mechanism:

(1) list: Provides a list of applications launched in response to requests received from the application manager **32**.

(2) jexec: Invokes the process cloning mechanism, parses command line arguments and executes a new instance of the master JVM process **33** as the cloned JVM process **34**. Preferably adopts a syntax compatible to standard JVM processes.

(3) kill: Terminates the application identified by an application handle or process identifier.

Other commands are possible, such as described in commonly-assigned U.S. patent application Ser. No. 10/745,164, entitled "System And Method For Performing Incremental Initialization Of A Master Runtime System Process," filed 22 Dec. 2003, pending, the disclosure of which is incorporated by reference.

During initialization, the master JVM process **33** also preloads classes **36** and classes defined in the class libraries **37** that are likely to be required by applications at runtime. The classes and interfaces are identified through profiling by ranking a set of classes according to a predetermined criteria, such as described in commonly-assigned U.S. patent application Ser. No. 09/970,661, filed Oct. 5, 2001, pending, the disclosure of which is incorporated by reference. A set of core Java foundation classes is specified in a bootstrap class loader **39** and application classes in a system application class loader **40**. Class loading requires identifying a binary form of a class type as identified by specific name, as further described below with reference to FIG. **10**. Depending upon whether the class was previously loaded or referenced, class loading can include retrieving a binary representation from source and constructing a class object to represent the class in memory. The master JVM process **33** maintains an internal symbol table (not shown) of classes previously loaded to resolve symbolic references. If the internal symbol table does not already contain an entry for the class name or class loader, the class loader responsible for loading the class is identified, invoked and given the name of the class.

The master JVM process **33** invokes the bootstrap class loader **39** and system application class loader **40** for every class likely to be requested by the applications. Thus, the prewarmed state **41** includes the class loading for applications prior to actual execution and the initialized and loaded classes are inherited by each cloned JVM process **34** as the inherited prewarmed state **42**.

Master JVM Process Mapping

FIG. **3** is a block diagram **60** showing, by way of example, a master JVM process **33** mapped into memory **62**. Generally, the context for an executing process includes a data space, user stack, kernel stack, and a user area that lists open files, current directory and supervisory permission settings. Other types of context can also be provided. The context is stored and managed in the memory **62** by the operating system. At device boot time, the operating system instantiates a representation of the executable master JVM process **33** into the memory **62**, possibly in non-contiguous pages **64***a-d*, and records the allocation of the memory space as page table entries **63** into the page table **61** prior to commencing execution of the master JVM process **33**. As well, the master JVM process context could similarly be mapped using other memory management systems, such as using demand paging, swapping and similar process memory allocation schemes compatible with process cloning, particularly process cloning with copy-on-write semantics.

Cloned JVM Process Mapping

FIG. **4** is a block diagram **70** showing, by way of example, a master JVM process **33** and a cloned JVM process **34** mapped into memory **62** through memory space cloning. In a system with process cloning that does not provide copy-on-write semantics, physical copies of the pages **64***a-c* in the memory **62** storing the parent process context are created for each child process. In response to a process cloning request, the operating system instantiates a copy of the representation of the executable master JVM process **33** for the cloned JVM process **34** into the memory **62**, possibly in non-contiguous pages **72***a-d*, and records the allocation of the memory space as page table entries **71** into the page table **61** prior to commencing execution of the cloned JVM process **34**. Thus, the cloned JVM process **34** is created with a physical copy of the context of the master JVM process **33**. Since a new, separate physical copy of the master JVM process context is created, the cloned JVM process **34** inherits the prewarmed state **41**, including the preloaded classes of the master JVM process **33**. However, the overall memory footprint of the runtime environment **31** is increased by the memory space required to store the additional copy of the master JVM process context.

Cloned JVM Process Mapping with Copy-On-Write

FIGS. **5**A-B are block diagrams **80**, **90** showing, by way of example, a master JVM process **33** and a cloned JVM process **34** mapped into memory **62** through memory space cloning with copy-on-write semantics. In a system with process cloning that provides copy-on-write semantics, only copies of the references, typically page table entries, to the memory space storing the parent process context are created for each child process. Referring first to FIG. **5**A, in response to a process cloning request, the operating system copies only the page table entries **63** referencing the memory space of the executable master JVM process **33** as a new set of page table entries **81** for the cloned JVM process **34**. Thus, the cloned JVM process **34** uses the same references to the possibly non-contiguous pages **64***a-d* storing the master JVM process context as the master JVM process **34**. Initialization and execution of the application associated with the cloned JVM process **34** requires less time, as only the page table entries **62** are copied to clone the master JVM process context. Furthermore, until the cloned JVM process **34** attempts to modify the master JVM process context, the memory space is treated as read only data, which can be shared by other processes.

Referring next to FIG. **5**B, the cloned JVM process **34** has attempted to modify one of the pages **82***c* in the memory space of the master JVM process context. In response, the operating system creates a physical copy of the to-be-modified memory space page **82***c* as a new page **91** and updates the allocation in the page table entries **81** for the cloned JVM process **34**. Through copy-on-write semantics, the overall footprint of the runtime environment **31** is maintained as small as possible and only grows until, and if, each cloned JVM process **34** actually requires additional memory space for application-specific context.

Method for Preloading Classes

FIG. **6** is a flow diagram, showing a method **100** for dynamic preloading of classes through memory space cloning of a master runtime system process, in accordance with the invention. The method **100** is described as a sequence of process operations or steps, which can be executed, for instance, by the runtime environment **31** of FIG. **2** or other components.

Initially, the application manager **32** is loaded (block **101**). The master JVM process **33** is loaded and initialized at device boot time (block **102**), as further described below with reference to FIG. **7**. Following loading and initialization, the master JVM process **33** enters an inactive sleep mode (block **103**). Upon receiving a request from the application manager **32** (block **104**), the master JVM process **33** awakens (block **105**). If necessary, the master JVM process **33** checks the network connection identifier (ID) (block **106**) for the application manager **32** and determines the type of request (block **107**). The master JVM process **33** recognizes the commands list, jexec, and kill, as described above with reference to FIG. **2**. If the request type corresponds to a jexec request, instructing the master JVM process **33** to initiate an execution of an application through process cloning (block **108**), a cloned JVM process **34** is cloned and executed (block **109**), as further described below with reference to FIGS. **8** and **9**. Processing continues indefinitely until the master JVM process **33** and the runtime environment **31** are terminated.

Routine for Loading Master JVM Process

FIG. **7** is a flow diagram showing the routine **120** for loading a master JVM process **33** for use in the method **100** of FIG. **6**. One purpose of the routine is to invoke the master JVM process **33** and to preload classes into the prewarmed state **41** for inheritance by cloned JVM processes **34**.

Initially, the master JVM process **33** begins execution at device boot time (block **121**). The master JVM process **33** then preloads classes as a part of the initialization process (block **122**), as further described below with reference to FIG. **10**. Briefly, preloading classes involves executing the bootstrap class loader **39** and system application class loader **40** to create and resolve classes likely required by one or more of the applications. The master JVM process **33** completes any other warmup operations (block **123**) and the routine returns.

Routine for Process Cloning without Copy-On-Write

FIG. **8** is a flow diagram showing the routine **130** for cloning a process without copy-on-write for use in the method **100** of FIG. **6**. One purpose of the routine is to create and initiate execution of a cloned JVM process **34** through process cloning that does not provide copy-on-write semantics.

Initially, the memory space containing the context of the master JVM process **33** is physically copied into a new memory space for the cloned JVM process **34** (block **131**). Optionally, the master JVM process **33** can set operating system level resource management parameters over the cloned JVM process **34** (block **132**), including setting scheduling priorities and limiting processor and memory consumption. Other types of resource management controls are possible. The cloned JVM process **34** is then executed by the

runtime environment **31** (block **133**) using the duplicated master JVM process context. The routine returns upon the completion (block **134**) of the cloned JVM process **34**.

Routine for Process Cloning with Copy-On-Write

FIG. **9** is a flow diagram showing the routine **140** for cloning a process with copy-on-write for use in the method **100** of FIG. **6**. One purpose of the routine is to create and initiate execution of a cloned JVM process **34** through process cloning that provides copy-on-write semantics.

Initially, references to the memory space containing the context of the master JVM process **33** are copied for the cloned JVM process **34** (block **141**). Optionally, the master JVM process **33** can set operating system level resource management parameters over the cloned JVM process **34** (block **142**), including setting scheduling priorities and limiting processor and memory consumption. Other types of resource management controls are possible. The cloned JVM process **34** is then executed by the runtime environment **31** (block **143**) using the referenced master JVM process context. Each time the cloned JVM process **34** attempts to write into the memory space referenced to the master JVM process context (block **144**), the operating system copies the applicable memory segment (block **145**). Otherwise, the cloned JVM process **34** continues to use the referenced master JVM process context (block **146**), which is treated as read only data. The routine returns upon the completion (block **147**) of the cloned JVM process **34**.

Routine for Preloading Class

FIG. **10** is a flow diagram showing the routine **150** for preloading a class **36** for use in the routine **120** of FIG. **7**. One purpose of the routine is to find and instantiate prewarmed instances of classes **36** and classes defined in the class libraries **37** as specified in the bootstrap class loader **39** and system application class loader **40** as prewarmed state **41** in the master JVM process **33** for inheritance by a cloned JVM process **34**.

Initially, the bootstrap class loader **39** and system application class loader **40** is located and invoked by the master JVM process **33** (block **151**). Each class **36** and class contained in a class library **37** is then iteratively processed (blocks **152**-**163**) as follows. First, the master JVM process **33** attempts to locate the class in a system class dictionary (block **153**). If the class is found (block **154**), no further class loading need be performed. Otherwise, the master JVM process **33** attempts to locate the class (block **155**) through standard Java class path location. If the class is found (block **156**), no further class loading need be performed. Otherwise, the master JVM process **33** attempts to load the bytes for the class from the source associated with the applicable bootstrap class loader **39** and system application class loader **40** (block **157**). If successful (block **158**), an instance of the class is created by compiling the source and the class instance is installed in the system class dictionary (block **160**). If the bytes for the class cannot be loaded from the source (block **158**), the master JVM process **33** throws a class not found exception (block **159**). Following the loading or attempted loading of the class, if the class requires resolution with respect to symbolic references (block **161**), the class is resolved by identifying the applicable class loader for the fully qualified class (block **162**). Processing continues with the next class (block **163**), after which the routine returns.

While the invention has been particularly shown and described as referenced to the embodiments thereof, those skilled in the art will understand that the foregoing and other changes in form and detail may be made therein without departing from the spirit and scope of the invention.

What is claimed is:

1. A system for dynamic preloading of classes through memory space cloning of a master runtime system process, comprising:

A processor; A memory a class preloader to obtain a representation of at least one class from a source definition provided as object-oriented program code;

a master runtime system process to interpret and to instantiate the representation as a class definition in a memory space of the master runtime system process;

a runtime environment to clone the memory space as a child runtime system process responsive to a process request and to execute the child runtime system process; and

a copy-on-write process cloning mechanism to instantiate the child runtime system process by copying references to the memory space of the master runtime system process into a separate memory space for the child runtime system process, and to defer copying of the memory space of the master runtime system process until the child runtime system process needs to modify the referenced memory space of the master runtime system process.

2. A system according to claim **1**, further comprising:

a cache checker to determine whether the instantiated class definition is available in a local cache associated with the master runtime system process.

3. A system according to claim **2**, further comprising:

a class locator to locate the source definition if the instantiated class definition is unavailable in the local cache.

4. A system according to claim **1**, further comprising:

a class resolver to resolve the class definition.

5. A system according to claim **1**, further comprising:

at least one of a local and remote file system to maintain the source definition as a class file.

6. A system according to claim **1**, further comprising:

a process cloning mechanism to instantiate the child runtime system process by copying the memory space of the master runtime system process into a separate memory space for the child runtime system process.

7. A system according to claim **1**, wherein the master runtime system process is caused to sleep relative to receiving the process request.

8. A system according to claim **1**, wherein the object-oriented program code is written in the Java programming language.

9. A system according to claim **8**, wherein the master runtime system process and the child runtime system process are Java virtual machines.

10. A method for dynamic preloading of classes through memory space cloning of a master runtime system process, comprising:

executing a master runtime system process;

obtaining a representation of at least one class from a source definition provided as object-oriented program code;

interpreting and instantiating the representation as a class definition in a memory space of the master runtime system process; and

cloning the memory space as a child runtime system process responsive to a process request and executing the child runtime system process;

wherein cloning the memory space as a child runtime system process involves instantiating the child runtime system process by copying references to the memory space of the master runtime system process into a separate memory space for the child runtime system process; and

wherein copying references to the memory space of the master runtime system process defers copying of the memory space of the master runtime system process until the child runtime system process needs to modify the referenced memory space of the master runtime system process.

11. A method according to claim **10**, further comprising:

determining whether the instantiated class definition is available in a local cache associated with the master runtime system process.

12. A method according to claim **11**, further comprising:

locating the source definition if the instantiated class definition is unavailable in the local cache.

13. A method according to claim **10**, further comprising:

resolving the class definition.

14. A method according to claim **10**, further comprising:

maintaining the source definition as a class file on at least one of a local and remote file system.

15. A method according to claim **10**, further comprising:

instantiating the child runtime system process by copying the memory space of the master runtime system process into a separate memory space for the child runtime system process.

16. A method according to claim **10**, further comprising:

causing the master runtime system process to sleep relative to receiving the process request.

17. A method according to claim **10**, wherein the object-oriented program code is written in the Java programming language.

18. A method according to claim **17**, wherein the master runtime system process and the child runtime system process are Java virtual machines.

19. A computer-readable storage medium holding code for performing the method according to claim **10**.

20. An apparatus for dynamic preloading of classes through memory space cloning of a master runtime system process, comprising:

A processor; A memory means for executing a master runtime system process;

means for obtaining a representation of at least one class from a source definition provided as object-oriented program code;

means for interpreting and means for instantiating the representation as a class definition in a memory space of the master runtime system process; and

means for cloning the memory space as a child runtime system process responsive to a process request and means for executing the child runtime system process;

wherein the means for cloning the memory space is configured to clone the memory space of a child runtime system process using a copy-on-write process cloning mechanism that instantiates the child runtime system process by copying references to the memory space of the master runtime system process into a separate memory space for the child runtime system process and that defers copying of the memory space of the master runtime system process until the child runtime system process needs to modify the referenced memory space of the master runtime system process.

21. A system according to claim **1**, further comprising:

a resource controller to set operating system level resource management parameters on the child runtime system process.

22. A method according to claim **10**, further comprising:

setting operating system level resource management parameters on the child runtime system process.

* * * * *

EXHIBIT E

US00RE38104E

(19) **United States**
(12) **Reissued Patent**
Gosling

(10) **Patent Number: US RE38,104 E**
(45) **Date of Reissued Patent: Apr. 29, 2003**

(54) **METHOD AND APPARATUS FOR RESOLVING DATA REFERENCES IN GENERATED CODE**

(75) Inventor: **James Gosling**, Redwood City, CA (US)

(73) Assignee: **Sun Microsystems, Inc.**, Palo Alto, CA (US)

(*) Notice: This patent is subject to a terminal disclaimer.

(21) Appl. No.: **09/261,970**

(22) Filed: **Mar. 3, 1999**

Related U.S. Patent Documents

Reissue of:

(64) Patent No.: **5,367,685**
Issued: **Nov. 22, 1994**
Appl. No.: **07/994,655**
Filed: **Dec. 22, 1992**

(51) **Int. Cl.**[7] **G06F 9/45**
(52) **U.S. Cl.** **717/140**; 717/106; 717/136; 717/139; 717/142; 717/146
(58) **Field of Search** 717/2, 5, 7, 8, 717/106–108, 114, 116, 146–147

(56) **References Cited**

U.S. PATENT DOCUMENTS

4,636,940	A	*	1/1987	Goodwin, Jr.	717/4
4,667,290	A		5/1987	Goss et al.	
4,667,920	A	*	5/1987	Goss et al.	248/610
4,686,623	A	*	8/1987	Wallace	717/8
4,729,096	A	*	3/1988	Larson	717/5
4,773,007	A	*	9/1988	Kanada et al.	717/9
5,201,050	A	*	4/1993	McKeeman et al.	717/7
5,230,050	A	*	7/1993	Iitsuka et al.	717/7
5,276,881	A		1/1994	Chan et al.	
5,280,613	A		1/1994	Chan et al.	
5,307,492	A	*	4/1994	Benson	717/7
5,313,614	A	*	5/1994	Goettelmann et al.	717/5
5,339,419	A		8/1994	Chan et al.	
5,347,632	A		9/1994	Filepp et al.	
5,428,792	A		6/1995	Conner et al.	

(List continued on next page.)

OTHER PUBLICATIONS

Adele Goldberg and David Robson, "Smalltalk–80–The Language and its Implementation", Xerox Palo Alto Research Center, 1983 (reprinted with corrections, Jul. 1985) pp. 1–720.

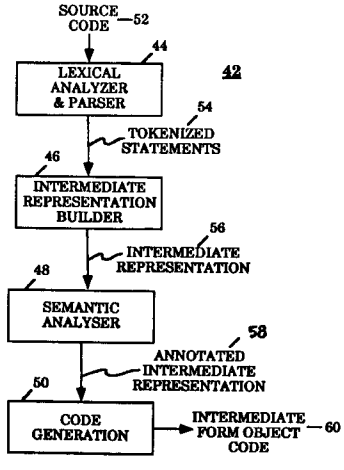
(List continued on next page.)

Primary Examiner—Thomas M. Heckler
(74) *Attorney, Agent, or Firm*—Finnegan, Henderson, Farabow, Garrett & Dunner, L.L.P.

(57) **ABSTRACT**

A hybrid compiler-interpreter comprising a compiler for "compiling" source program code, and an interpreter for interpreting the "compiled" code, is provided to a computer system. The compiler comprises a code generator that generates code in intermediate form with data references made on a symbolic basis. The interpreter comprises a main interpretation routine, and two data reference handling routines, a dynamic field reference routine for handling symbolic references, and a static field reference routine for handling numeric references. The dynamic field reference routine, when invoked, resolves a symbolic reference and rewrites the symbolic reference into a numeric reference. After re-writing, the dynamic field reference routine returns to the main interpretation routine without advancing program execution to the next instruction, thereby allowing the rewritten instruction with numeric reference to be reexecuted. The static field reference routine, when invoked, obtain data for the program from a data object based on the numeric reference. After obtaining data, the static field reference routine advances program execution to the next instruction before returning to the interpretation routine. The main interpretation routine selectively invokes the two data reference handling routines depending on whether the data reference in an interaction in a symbolic or a numeric reference.

31 Claims, 5 Drawing Sheets



U.S. PATENT DOCUMENTS

5,442,771	A		8/1995	Filepp et al.	
5,594,910	A		1/1997	Filepp et al.	
5,613,117	A	*	3/1997	Davidson et al.	717/8
5,649,204	A		7/1997	Pickett	
5,758,072	A		5/1998	Filepp et al.	
5,836,014	A	*	11/1998	Faiman, Jr.	717/7

OTHER PUBLICATIONS

Andrew Black, Norman Hutchinson, Eric Jul and Henry Levy, "Distribution and Abstract Types in Emerald", University of Washington, Technical Report No. 85–08–05, Aug. 1985, pp. 1–10.

Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy, "Object Structure in the Emerald System", University of Washington, Technical Report 86–04–03, Apr. 1986, pp. 1–14.

Andrew Blaine Proudfoot, "Replects: data replication in the Eden System", Department of Computer Science, University of Washington, Technical Report No. TR–85–12–04, Dec. 1985, pp. 1–156.

Andrew P. Black and Henry M. Levy, "A Language for Distributed Programming", Department of Computer Science, University of Washington, Technical Report 86–02–03, Feb. 1986, p. 10.

Andrew P. Black, "Supporting Distributed Applications: Experience with Eden", Department of Computer Science, University of Washington, Technical Report 85–03–02, Mar. 1985, pp. 1–21.

Andrew P. Black, "The Eden Programming Language", Department of Computer Science, FR–35, University of Washington, Technical Report 85–09–01, Sep. 1985 (Revised, Dec. 1985), pp. 1–19.

Andrew P. Black, "The Eden Project: Overview and Experiences", Department of Computer Science, University of Washington, EUUG, Autumn '86 Conference Proceedings, Manchester, UK, Sep. 22–25 1986, pp. 177–189.

Andrew P. Black, Edward D. Lazowska, Jerre D. Noe and Jan Sanislo, "The Eden Project: A Final Report", Department of Computer Science, University of Washington, Technical Report 86–11–01, Nov. 1986, pp. 1–28.

Calton Pu, "Replication and Nested Transactions in the Eden Distributed System", Doctoral Disseration, University of Washington, Aug. 6, 1986, pp. 1–179 (1 page Vita).

Cara Holman and Guy Almes, "The Eden Shared Calendar System", Department of Computer Science, FR–35, University of Washington, Technical Report 85–05–02, Jun. 22, 1985, pp. 1–14.

Eric Jul, "Object Mobility in a Distributed Object–Oriented System", a Dissertation, University of Washington, 1989, pp. 1–154 (1 page Vita).

Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black, "Fine–Grained Mobility in the Emerald System", University of Washington, ACM Transactions on Computer Systems, vol. 6, No. 1, Feb. 1988, pp. 109–133.

Felix Samson Hsu, "Reimplementing Remote Procedure Calls", University of Washington, Thesis Mar. 22, 1985, pp. 1–106.

Guy Almes, Andrew Black, Carl Bunje and Douglas Wiebe, "Edmas: A locally Distributed Mail System", Department of Computer Science, University of Washington, Technical Report 83–87–01, Jul. 7, 1983, Abstract & pp. 1–17.

Guy T. Almes, "Integration and Distribution in the Eden System", Department of Computer Science, University of Washington, Technical Report 83–01–02, Jan. 19, 1983, pp. 1–18 & Abstract.

Guy T. Almes, "The Evolution of th Eden Invocation Mechanism", Department of Computer Science, University of Washington, Technical Report 83–01–03, Jan. 19, 1983, pp. 1–14 & Abstract.

Guy T. Almes, Andrew P. Black, Edward D. Lazawska, and Jerre D. Noe, "The Eden System: A Technical Review", Department of Computer Science, University of Washington, Technical Report 83–10–05, Oct. 1983, pp. 1–25.

Guy T. Almes, Michael J. Fischer, Hellmut Golde, Edward D. Lazawska, Jerre D. Noe, "Research in Integrated Distributed Computing", Department of Computer Science, University of Washington, Oct. 1979, pp. 1–42.

Krasner et al., "Smalltalk–80: Bits of History, Words of Advice", 1983 Xerox Corporation, pp. 1–344.

Norman C. Hutchinson, "Emerald: An Object–Based Language for Distributed Programming", a Dissertation, University of Washington, 1987, pp. 1–107.

Proceedings of the Eighth Symposium on Operating Systems Principles, Dec. 14–16, 1981, ACM, Special Interest Group on Operating Systems, Association for Computing Machinery, vol. 15, No. 5, Dec. 1981, ACM Order No. 534810.

Wm. A. Wulf, "PQCC: A Machine–Relative Compiler Technology," Carnegie–Mellon University, Pittsburgh, PA, Sep. 1980, pp. 1–22.

Inder–jeet S. Gujral, "Retargetable Code Generation for ADA* Compilers", SoftTech, Inc., Waltham, MA, Dec., 1981, pp. 1–13.

Nori et al., "The Pascal <P> Compiler: Implementation Notes", Jul. 1976, pp. 1–53.

Glanville et al., "A New Method for Compiler Code Generation (Extended Abstract)", Computer Science Division, University of California, Berkeley, CA, pp. 231–240.

Colusa Software White Paper: "Omniware Technical Overview", Colusa Software, Inc., 1995, pp. 1–14.

Colusa Software White Paper: Omniware: A Universal Substrate for Mobile Code: Colusa Software, Inc., pp. 1–13.

Ali–Reza Adl–Tabatabai et al., "Efficient and Language–Independent Mobile Programs", Proceedings of PLDI '96, ACM SIGPLAN '96 Conf. on Programming Language Design and Implementation, May, 1996, pp. 1–10.

Lucco et al., "Omniware: A Universal Substrate for Web Programming", pp. 1–11.

Wahbe et al., "Efficient Software–Based Fault Isolation", Computer Science Division, University of California, Berkeley, CA, pp. 203–216.

Graham et al., "Adaptable Binary Programs", 1995 Usenix Technical Conference—Jan., 1995, New Orleans, LA, pp. 315–325.

Steven Lucco, "High–Performance Microkernel Systems", School of Computer Science, Carengie Mellon University. p. 1.

Sawdon et al., "A Preliminary Report on Software Prefetching in the Instruction Stream", School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, pp. 1–7.

Bolosky, et al., "Operating System Directions for the Next Millennium", Microsoft Research, Redmond, WA, pp. 1–7.

1995 Project Summaries: "Software System Support for High Performance Multicomputing", School of Computer Science, Carnegie Mellon University, Jul. 1995, pp. 1–4.

Ernst et al., "Code Compression", 1997, pp. 358–365.
Peter Deutsch et al., "Efficient Implementation of the Smalltalk–80 System", 1983, pp. 297–302.
Engelstad et al., "A Dynamic C–Based Object–Oriented System for UNIX", IEEE Software, May, 1991, pp. 73–85.
Gerring, et al., "S–1 U–Code, A Universal P–Code for the S–1 Project (PAIL–6)", Stanford University, Computer Science Department, Technical Note No. 159, Aug., 1979, pp. 1–7.
Gary McWilliams, "Digital's Architectural Gamble", Datamation, Mar., 1989, pp. 14–24.
"Architecture–Neutral Distribution Format", Open Software Foundation, Cambridge, MA, pp. 1–3.
Wolf et al., "Portable Compiler Eases Problems of Software Migration", System Design/Software, pp. 147–153.
Fischer et al., "Crafting a Compiler", 1988, pp. 551–555, 632–641.
Anklam et al., "Engineering a Compiler, VAX–11 Code Generation and Optimization", 1982 Digital Equipment Corporation, pp. 124–137.
Tanenbaum et al., "A Practical Tool Kit for Making Portable Compilers", Computing Practices, Communications of the ACM, Sep., 1983, vol. 26, No. 9, pp. 654–660.
Almasi et al., "Highly Parallel Computing", pp. 247–277.
Ann Sussman, "OSF Eyes Shrink–Wrap RFT", Unix Today, pp. 1, 43.
Evan Grossman, "OSF Adds Ingredients to Operating System", PC Week, Mar. 27, 1989.
Sites et al., Universal P–Code Definition, Version 0.3, Department of Electrical Engineering and Computer Sciences, University of California at San Diego, Jul. 1979, pp. 5–9.
Goldberg et al., "Smalltalk–80: The Language and Its Implementation", Addison–Wesley, Reading, MA, 1983, pp. 594–598.
Richard L. Sites and Daniel R. Perkins, "Universal P–Code Definition, version (0.3)," Dept. of Electrical Engineering and Computer Sciences, University of California at San Diego, Jul., 1979, pp. 1–40.
Richard L. Sites et al., "Machine–Independent Pascal Optimizer Project," UCSD/CS–79/038, Nov. 1979, pp. 1–94.
Peter Nye, U–CODE: An Intermediate Language for Pascal and Fortran (PAIL–8), Feb. 16, 1980, pp. 1–37–2.
Chung, Kin–Man and Yuen, Herbert, "A 'Tiny' Pascal Compiler: the P–Code Interpreter," BYTE Publications, Inc., Sep. 1978.
Chung, Kin–Man and Yuen, Herbert, "A 'Tiny' Pascal Compiler: Part 2: The P–Compiler," BYTE Publications, Inc., Oct. 1978.
Thompson, Ken, "Regular Expression Search Algorithm," Communications of the ACM, vol. II, No. 6, p. 149 et seq., Jun. 1968.
Mitchell, James G., Maybury, William, and Sweet, Richard, Mesa Language Manual, Xerox Corporation.
McDaniel, Gene, "An Analysis of a Mesa Instruction Set," Xerox Corporation, May 1982.
Pier, Kenneth A., "A Retrospective on the Dorado, A High–Performance Personal Computer," Xerox Corporation, Aug. 1983.
Pier, Kenneth A., "A Retrospective on the Dorado, A High–Performance Personal Computer," IEEE Conference Proceedings, The 10th Annual International Symposium on Computer Architecture, 1983.
Goldberg, Adele and Robson, David, "Smalltalk—80: The Language," ParcPlace Systems and Xerox PARC, Addison–Wesley Publishing Company, 1989, Chap. 21, pp. 417–442.
Budd, Timothy, "A Little Smalltalk," Oregon State University, Addison–Wesley Publishing Company, 1987, Chap. 13, pp. 150–160, Chapter 14, pp. 161–175, Chapter 15, pp. 176–192.
Krasner, Glenn, "The Smalltalk–80 Virtual Machine" BYTE Publications Inc., Aug. 1991, pp. 300–320.
Engelstad, Steve, et al., "A Dynamic C–Based Object–Oriented System for Unix," Software, May 1991, pp. 73–85.
Deutsch, L. Peter, et al., "Efficient Implementation of the Smartalk–80 System," Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages, Jan. 15–18, 1984, pp. 297–302.

* cited by examiner

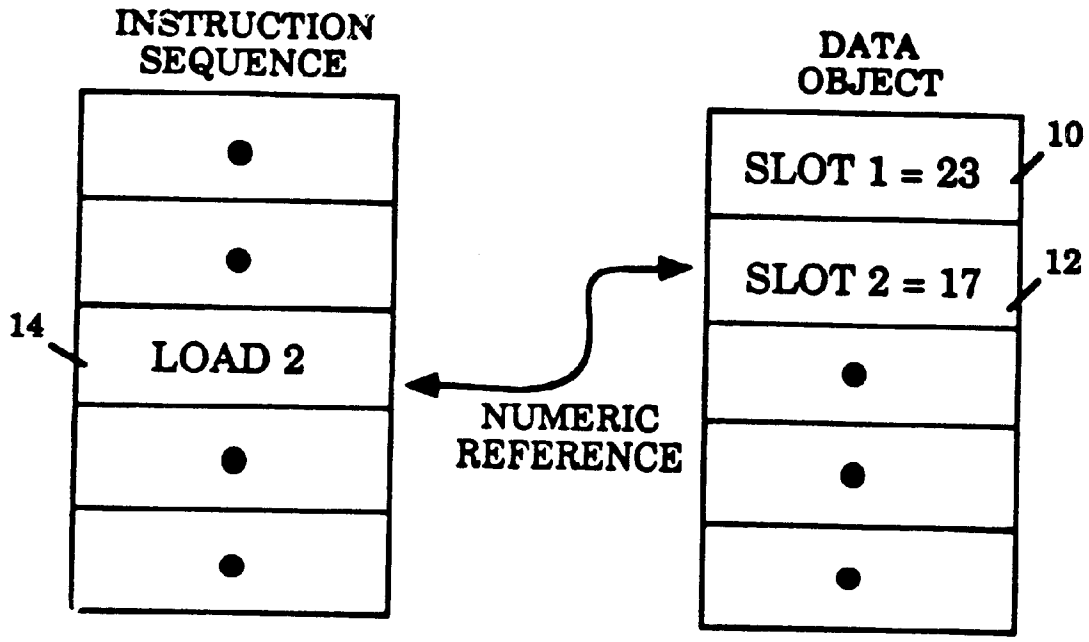


Figure 1A

Prior Art

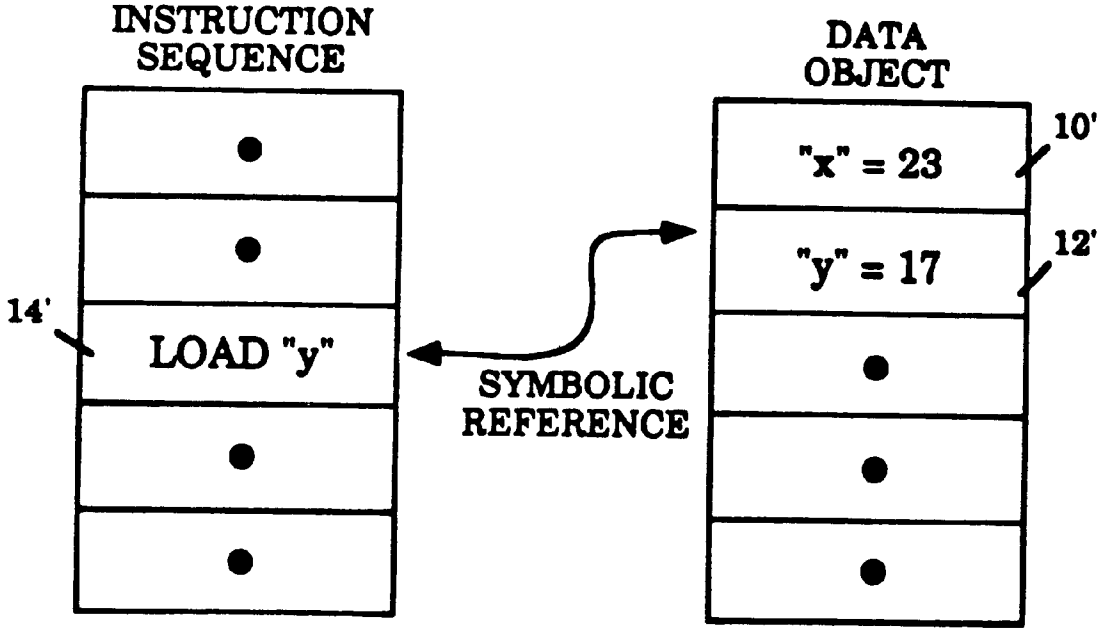


Figure 1B

Prior Art

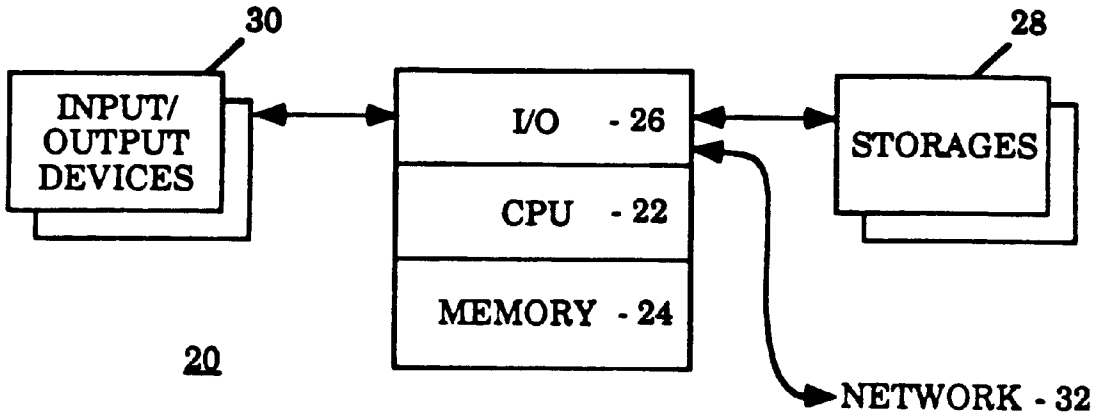


Figure 2

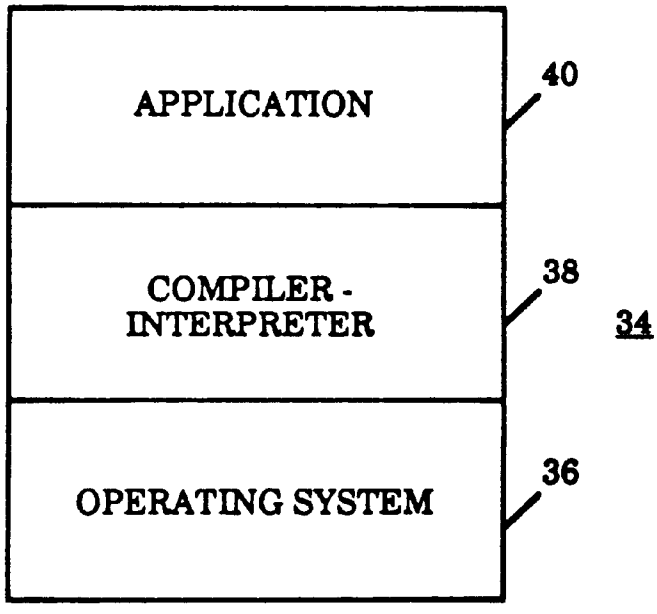


Figure 3

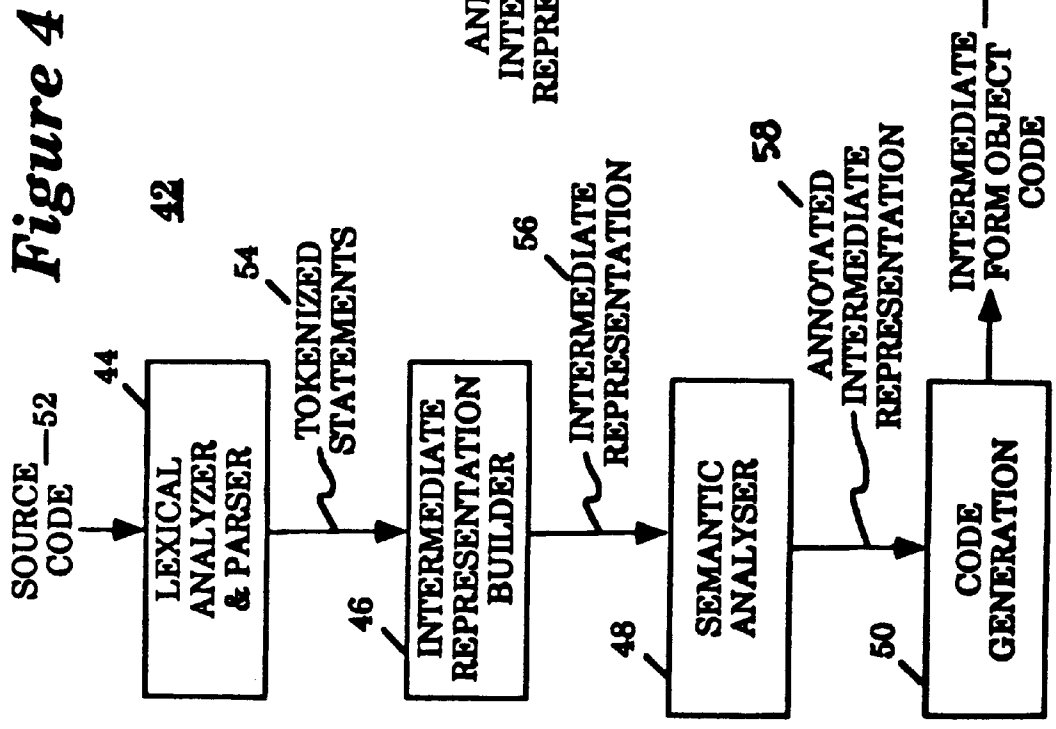


Figure 4

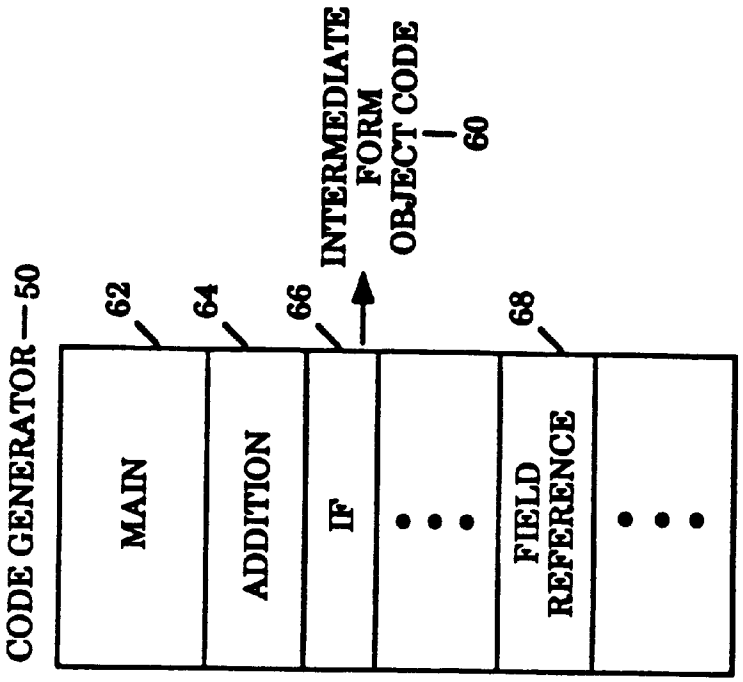


Figure 5

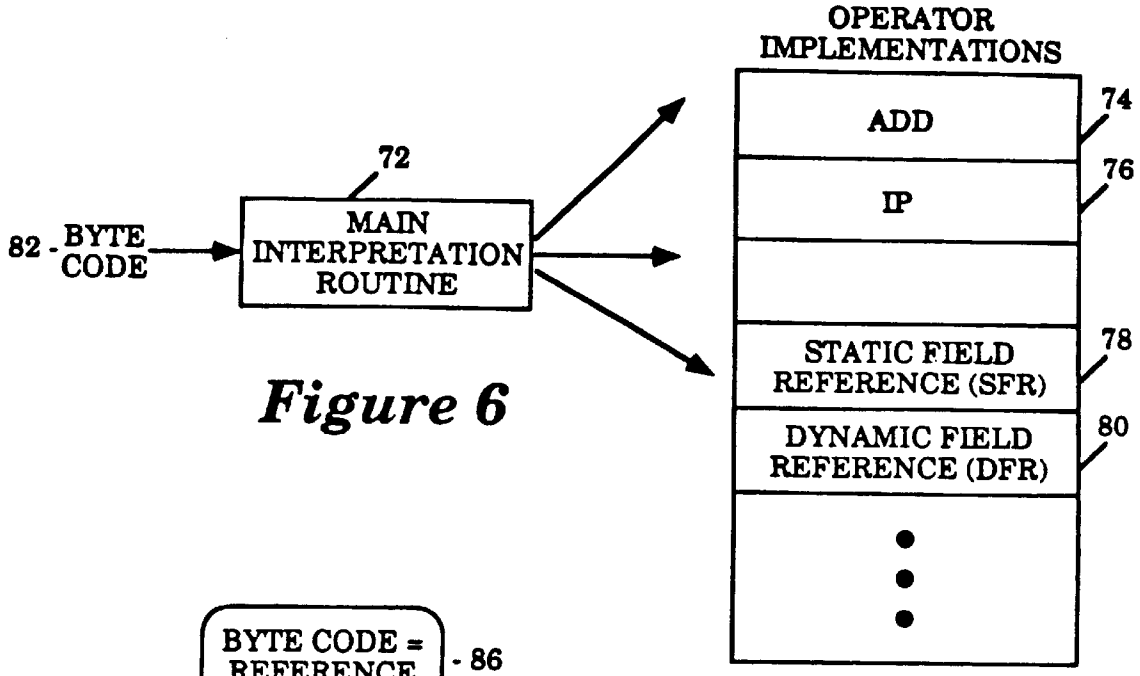


Figure 6

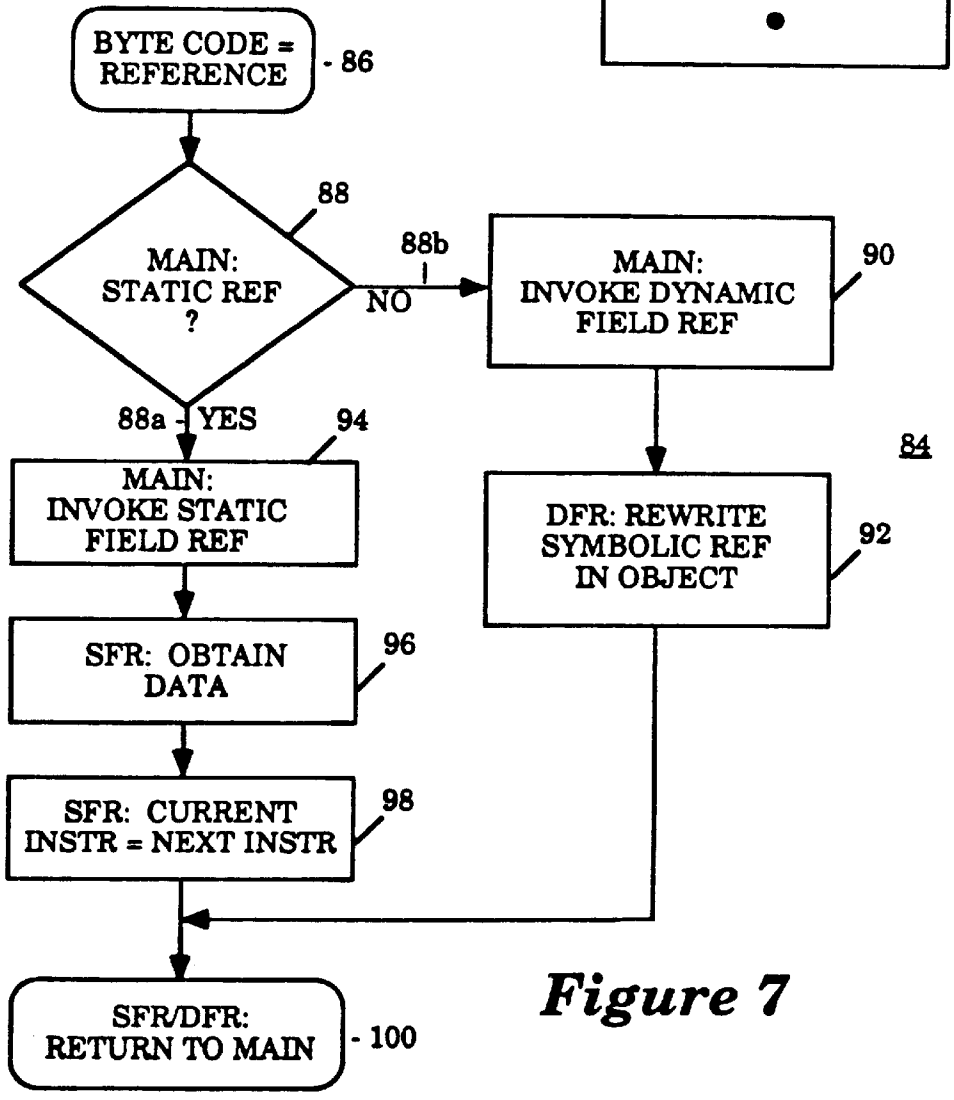


Figure 7

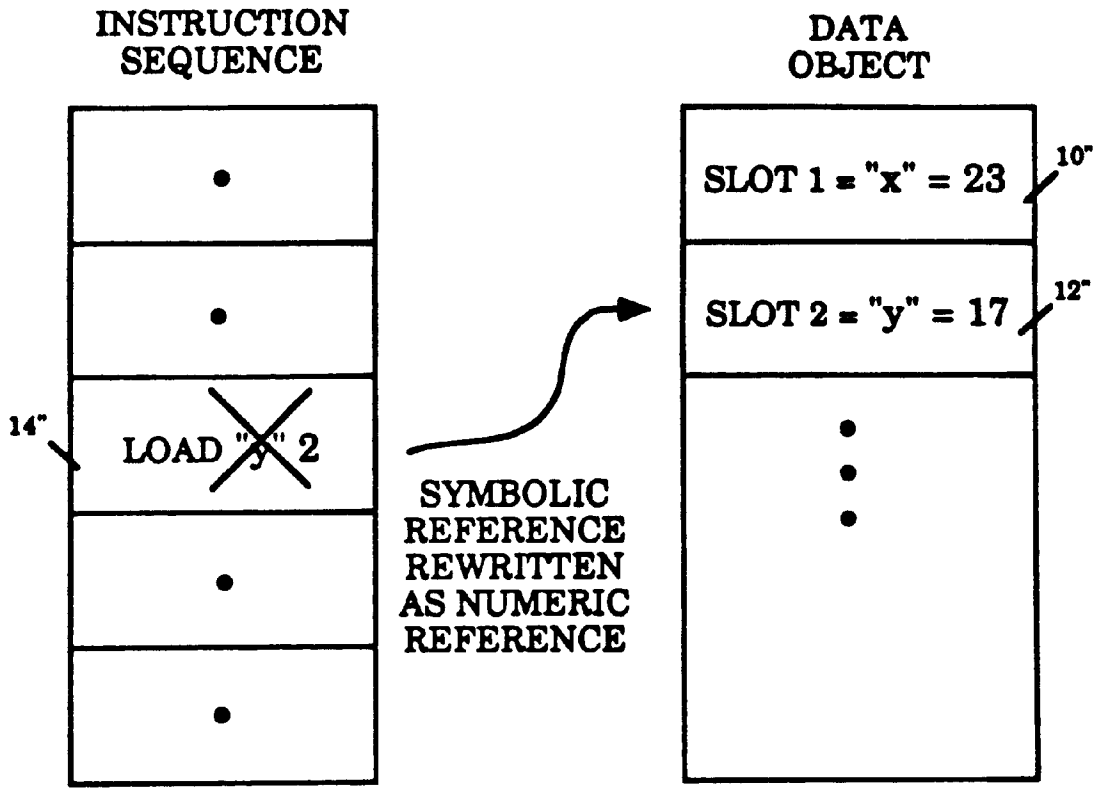

INSTRUCTION SEQUENCE
DATA OBJECT
SLOT 1 = "x" = 23
10"
SLOT 2 = "y" = 17
12"
14"
LOAD "y" 2
SYMBOLIC REFERENCE REWRITTEN AS NUMERIC REFERENCE

Figure 8

METHOD AND APPARATUS FOR RESOLVING DATA REFERENCES IN GENERATED CODE

Matter enclosed in heavy brackets [ ] appears in the original patent but forms no part of this reissue specification; matter printed in italics indicates the additions made by reissue.

This is a continuation of reissue application Ser. No. 08/755,764, filed Nov. 21, 1996, now U.S. Pat. Re. No. 36,204, which is incorporated herein by reference.

BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention relates to the field of computer systems, in particular, programming language compilers and interpreters of these computer systems. More specifically, the present invention relates to resolving references in compiler generated object code.

2. Background

The implementation of modern programming languages, including object oriented programming languages, are generally grouped into two categories: compiled and interpreted.

In a compiled programming language, a computer program (called a compiler) compiles the source program and generates executable code for a specific computer architecture. References to data in the generated code are resolved prior to execution based on the layout of the data objects that the program deals with, thereby, allowing the executable code to reference data by their locations. For example, consider a program that deals with a point data object containing two variables x and y, representing the x and y coordinates of a point, and further assume that the variables x and y are assigned slots **1** and **2** respectively, in each instance of the point data object. Thus, an instruction that accesses or fetches y, such as the Load instruction **14** illustrated in FIG. **1**, is resolved to reference the variable y by the assigned slot **2** before the instruction sequence is executed. Particular examples of programming language compilers that generate code and resolve data references in the manner described above include C and C++ compilers.

This "compiled" approach presents problems when a program is constructed in pieces, which happens frequently under object oriented programming. For example, a program may be constructed from a library and a main program. If a change is made to the library, such that the layout of one of the data objects it implements is changed, then clients of that library, like the main program, need to be recompiled. Continuing the preceding example, if the point data object had a new field added at the beginning called name, which contains the name of the point, then the variables x and y could be reassigned to slots **2** and **3**. Existing programs compiled assuming that the variables x and y and are in slots **1** and **2** will have to be recompiled for them to execute correctly.

In an interpreted language, a computer program (called a translator) translates the source statements of a program into some intermediate form, typically independent of any computer instruction set. References to data in the intermediate form are not fully resolved before execution based on the layout of the data objects that the program deals with. Instead, references to data are made on a symbolic basis. Thus, an instruction that accesses or fetches y, such as the Load instruction **14'** illustrated in FIG. **1**, references the variable y by the symbolic name "y". The program in

intermediate form is executed by another program (called an interpreter) which scans through the code in intermediate form, and performs the indicated actions. Each of the symbolic references is resolved during execution each time the instruction comprising the symbolic reference is interpreted. A particular example of a programming language interpreter that translates source code into intermediate form code and references data in the manner described above is the BASIC interpreter.

The "interpreted" approach avoids the problems encountered with the "compiled" approach, when a program is constructed in pieces. However, because of the extra level of interpretation at execution time, each time an instruction comprising a symbolic reference is interpreted, execution is slowed significantly.

Thus, it is desirable if programming languages can be implemented in a manner that provides the execution performance of the "compiled" approach, and at the same time, the flexibility of the "interpreted" approach for altering data objects, without requiring the compiled programs to be recompiled. As will be disclosed, the present invention provides a method and apparatus for resolving data references in compiler generated object code that achieves the desired results.

SUMMARY OF THE INVENTION

A method and apparatus for generating executable code and resolving data references in the generated code is disclosed. The method and apparatus provides execution performance substantially similar to the traditional compiled approach, as well as the flexibility of altering data objects like the traditional interpreted approach. The method and apparatus has particular application to implementing object oriented programming languages in computer systems.

Under the present invention, a hybrid compiler-interpreter comprising a compiler for "compiling" source program code, and an interpreter for interpreting the "compiled" code, is provided to a computer system. The compiler comprises a code generator that generates code in intermediate form with data references made on a symbolic basis. The interpreter comprises a main interpretation routine, and two data reference handling routines, a static field reference routine for handling numeric references and a dynamic field reference routine for handling symbolic references. The dynamic field reference routine, when invoked, resolves a symbolic reference and rewrites the symbolic reference into a numeric reference. After rewriting, the dynamic field reference routine returns to the interpreter without advancing program execution to the next instruction, thereby allowing the rewritten instruction with numeric reference to be reexecuted. The static field reference routine, when invoked, obtain data for the program from a data object based on the numeric reference. After obtaining data, the static field reference routine advances program execution to the next instruction before returning to the interpreter. The main interpretation routine selectively invokes the two data reference handling routines depending on whether the data reference in an instruction is a symbolic or a numeric reference.

As a result, the "compiled" intermediate form object code of a program achieves execution performance substantially similar to that of the traditional compiled object code, and yet it has the flexibility of not having to be recompiled when the data objects it deals with are altered like that of the traditional translated code, since data reference resolution is performed at the first execution of a generated instruction comprising a data reference.

BRIEF DESCRIPTION OF THE DRAWINGS

The objects, features, and advantages of the present invention will be apparent from the following detailed description of the presently preferred and alternate embodiments of the invention with references to the drawings in which:

FIG. **1** shows the prior art compiled approach and the prior art interpreted approach to resolving data reference.

FIG. **2** illustrates an exemplary computer system incorporated with the teachings of the present invention.

FIG. **3** illustrates the software elements of the exemplary computer system of FIG. **2**.

FIG. **4** illustrates one embodiment of the compiler of the hybrid compiler-interpreter of the present invention.

FIG. **5** illustrates one embodiment of the code generator of the compiler of FIG. **4**.

FIG. **6** illustrates one embodiment of the interpreter and operator implementations of the hybrid compiler-interpreter of the present invention.

FIG. **7** illustrates the cooperative operation flows of the main interpretation routine, the static field reference routine and the dynamic field reference routine of the present invention.

FIG. **8** illustrates an exemplary resolution and rewriting of a data reference under the present invention.

DETAILED DESCRIPTION PRESENTLY PREFERRED AND ALTERNATE EMBODIMENTS

A method and apparatus for generating executable code and resolving data references in the generated code is disclosed. The method and apparatus provides execution performance substantially similar to the traditional compiled approach, as well as the flexibility of altering data objects like the traditional interpreted approach. The method and apparatus has particular application to implementing object oriented programming languages. In the following description for purposes of explanation, specific numbers, materials and configurations are set forth in order to provide a thorough understanding of the present invention. However, it will be apparent to one skilled in the art that the present invention may be practiced without the specific details. In other instances, well known systems are shown in diagrammatical or block diagram form in order not to obscure the present invention unnecessarily.

Referring now to FIGS. **2** and **3**, two block diagrams illustrating an exemplary computer system incorporated with the teachings of the present invention are shown. As shown in FIG. **2**, the exemplary computer system **20** comprises a central processing unit (CPU) **22**, a memory **24**, and an I/O module **26**. Additionally, the exemplary computer system **20** also comprises a number of input/output devices **30** and a number of storage devices **28**. The CPU **22** is coupled to the memory **24** and the I/O module **26**. The input/output devices **30**, and the storage devices **28** are also coupled to the I/O module **26**. The I/O module **26** in turn is coupled to a network **32**.

Except for the manner they are used to practice the present invention, the CPU **22**, the memory **24**, the I/O module **26**, the input/output devices **30**, and the storage devices **28**, are intended to represent a broad category of these hardware elements found in most computer systems. The constitutions and basic functions of these elements are well known and will not be further described here.

As shown in FIG. **3**, the software elements of the exemplary computer system of FIG. **2** comprises an operating system **36**, a hybrid compiler-interpreter **38** incorporated with the teachings of the present invention, and applications compiled and interpreted using the hybrid compiler-interpreter **38**. The operating system **36** and t he applications **40** are intended to represent a broad categories of these software elements found in many computer systems. The constitutions and basic functions of these elements are also well known and will not be described further. The hybrid compiler-interpreter **38** will be described in further detail below with references to the remaining figures.

Referring now to FIGS. **4** and **5**, two block diagrams illustrating the compiler of the hybrid compiler-interpreter of the present invention are shown. Shown in FIG. **4** is one embodiment of the compiler **42** comprising a lexical analyzer and parser **44**, an intermediate representation builder **46**, a semantic analyzer **48**, and a code generator **50**. These elements are sequentially coupled to each other. Together, they transform program source code **52** into tokenized statements **54**, intermediate representations **56**, annotated intermediate representations **58**, and ultimately intermediate form code **60** with data references made on a symbolic basis. The lexical analyzer and parser **44**, the intermediate representation builder **46**, and the semantic analyzer **48**, are intended to represent a broad category of these elements found in most compilers. The constitutions and basic functions of these elements are well known and will not be otherwise described further here. Similarly, a variety of well known tokens, intermediate representations, annotations, and intermediate forms may also be used to practice the present invention.

As shown in FIG. **5**, the code generator **50** comprises a main code generation routine **62**, a number of complimentary operator specific code generation routines for handling the various operators, such as the ADD and the IF code generation routines, **64** and **66**, and a data reference handling routine **68**. Except for the fact that generated coded **60** are in intermediate form and the data references in the generated code are made on a symbolic basis, the main code generation routine **62**, the operator specific code generation routines, e.g. **64** and **66**, and the data reference handling routine **68**, are intended to represent a broad category of these elements found in most compilers. The constitutions and basic functions of these elements are well known and will not be otherwise described further here.

For further descriptions on various parsers, intermediate representation builders, semantic analyzers, and code generators, see A. V. Aho, R. Sethi, and J. D. Ullman, Compilers Principles, Techniques and Tools. Addision-Wesley, 1986, pp. 25–388, and 463–512.

Referring now to FIGS. **6** and **7**, two block diagrams illustrating one embodiment of the interpreter of the hybrid compiler-interpreter of the present invention and its operation flow for handling data references is shown. As shown in FIG. **6**, the interpreter **70** comprises a main interpretation routine **72**, a number of complimentary operator specific interpretations routines, such as the ADD and the IF interpretation routines, **74** and **76**, and two data reference interpretation routines, a static field reference routine (SFR) and a dynamic field reference routine (DFR), **78** and **80**. The main interpreter routine **72** receives the byte codes **82** of the intermediate form object code as inputs, and interprets them, invoking the operator specific interpretations routines, e.g. **74** and **76**, and the data reference routines, **78** and **80**, as necessary. Except for the dynamic field reference routine **80**, and the manner in which the main interpretation routine **72**

and the state field reference routine **78** cooperates with the dynamic field reference routine **80** to handle data references, the main interpretation routine **72**, the operator specific interpretation routines, e.g. **74** and **76**, and the static field reference routine **78**, are intended to represent a broad category of these elements found in most compilers and interpreters. The constitutions and basic functions of these elements are well known and will not be otherwise described further here.

As shown in FIG. **7**, upon receiving a data reference byte code, block **86**, the main interpretation routine determines if the data reference is static, i.e. numeric, or dynamic, i.e. symbolic, block **88**. If the data reference is a symbolic reference, branch **88**b, the main interpretation routine invokes the dynamic field reference routine, block **90**. Upon invocation, the dynamic field reference routine resolves the symbolic reference, and rewrites the symbolic reference in the intermediate form object code as a numeric reference, block **92**. Upon rewriting the data reference in the object code, the dynamic field reference routine returns to the main interpretation routine, block **100**, without advancing the program counter. As a result, the instruction with the rewritten numeric data reference gets reexecuted again.

On the other hand, if the data reference is determined to be a numeric reference, branch **88**a, the main interpretation routine invokes the static field reference routine, block **94**. Upon invocation, the static field reference routine obtain the data reference by the numeric reference, block **96**. Upon obtaining the data, the static field reference routine advances the program counter, block **98**, and returns to the main interpretation routine, block **100**.

Referring now to FIG. **8**, a block diagram illustrating the alteration and rewriting of data references under the present invention in further detail is shown. As illustrated, a data referencing instruction, such as the LOAD instruction **14**", is initially generated with a symbolic reference, e.g. "y". Upon its first interpretation in execution, the data referencing instruction, e.g. **14**, is dynamically resolved and rewritten with a numeric reference, e.g. slot **2**. Thus, except for the first execution, the extra level of interpretation to resolve the symbolic reference is no longer necessary. Therefore, under the present invention, the "compiled" intermediate form object code of a program achieves execution performance substantially similar to that of the traditional compiled object code, and yet it has the flexibility of not having to be recompiled when the data objects it deals with are altered like that of the traditional translated code, since data reference resolution is performed at the first execution of a generated instruction comprising a symbolic reference.

While the present invention has been described in terms of presently preferred and alternate embodiments, those skilled in the art will recognize that the invention is not limited to the embodiments described. The method and apparatus of the present invention can be practiced with modification and alteration within the spirit and scope of the appended claims. The description is thus to be regarded as illustrative instead of limiting on the present invention.

What is claimed is:

[1. In a computer system comprising a program in source code form, a method for generating executable code for said program and resolving data references in said generated code, said method comprising the steps of:

a) generating executable code in intermediate form for said program in source code form with data references being made in said generated code on a symbolic basis, said generated code comprising a plurality of instructions of said computer system;

b) interpreting said instructions, one at a time, in accordance to a program execution control;

c) resolving said symbolic references to corresponding numeric references, replacing said symbolic references with their corresponding numeric references, and continuing interpretation without advancing program execution, as said symbolic references are encountered while said instructions are being interpreted; and

d) obtaining data in accordance to said numeric references, and continuing interpretation after advancing program execution, as said numeric references are encountered while said instruction are being interpreted;

said steps b) through d) being performed iteratively and interleaving.**]**

[2. The method as set forth in claim **1**, wherein, said program in source code form is implemented in source code form of an object oriented programming language.**]**

[3. The method as set forth in claim **2**, wherein said programming language is C.**]**

[4. The method as set forth in claim **2**, wherein, said programming language is C++.**]**

[5. The method as set forth in claim **1**, wherein, said program execution control is a program counter said continuing interpretation in step c) is achieved by performing said step b) after said c) without incrementing said program counter; and

said continuing interpretation in said step d) is achieved by performing said step b) after said d) after incrementing said program counter.**]**

[6. In a computer system comprising a program in source code form, an apparatus for generating executable code for said program and resolving data references in said generated code, said apparatus comprising:

a) compilation means for receiving said program in source code form and generating executable code in intermediate form for said program in source code form with data references being made in said generated code on a symbolic basis, said generated code comprising a plurality of instructions of said computer system;

b) interpretation means for receiving said generated code and interpreting said instructions, one at a time;

c) dynamic reference handling means coupled to said interpretation means for resolving said symbolic references to corresponding numeric references, replacing said symbolic references with their corresponding numeric references, and continuing interpretation by said interpretation means without advancing program execution, as said symbolic references are encountered while said instructions are being interpreted by said interpretation means; and

d) static reference handling means coupled to said interpretation means for obtaining data in accordance to said numeric references, and continuing interpretation by said interpretation means after advancing program execution, as said numeric references are encountered while said instruction are being interpreted by said interpretation means;

said interpretation means, said dynamic reference handling means, and said static reference handling means performing their corresponding functions iteratively and interleavingly.**]**

[7. The apparatus as set forth in claim **6**, wherein, said program in source code form is implemented in source code form of an object oriented programming language.**]**

[8. The apparatus as set forth in claim **7**, wherein, said programming language is C.**]**

[**9**. The apparatus as set forth in claim **7**, wherein, said programming language is C++.]

[**10**. The apparatus as set forth in claim **6**, wherein, and program execution control is a program counter.]

11. An apparatus comprising:

a memory containing intermediate form object code constituted by a set of instructions, certain of said instructions containing one or more symbolic references; and

a processor configured to execute said instructions containing one or more symbolic references by determining a numerical reference corresponding to said symbolic reference, storing said numerical references, and obtaining data in accordance to said numerical references.

12. A computer-readable medium containing instructions for controlling a data processing system to perform a method for interpreting intermediate form object code comprised of instructions, certain of said instructions containing one or more symbolic references, said method comprising the steps of:

interpreting said instructions in accordance with a program execution control; and

resolving a symbolic reference in an instruction being interpreted, said step of resolving said symbolic reference including the substeps of:

determining a numerical reference corresponding to said symbolic reference, and

storing said numerical reference in a memory.

13. A computer-implemented method for executing instructions, certain of said instructions containing one or more symbolic references, said method comprising the steps of:

resolving a symbolic reference in an instruction, said step of resolving said symbolic reference including the substeps of:

determining a numerical reference corresponding to said symbolic reference, and

storing said numerical reference in a memory.

14. The method of claim 3, wherein said substep of storing said numerical reference comprises the substep of replacing said symbolic reference with said numerical reference.

15. The method of claim 3, wherein said step of resolving said symbolic reference further comprises the substep of executing said instruction containing said symbolic reference using the stored numerical reference.

16. The method of claim 3, wherein said step of resolving said symbolic reference further comprises the substep of advancing program execution control after said substep of executing said instruction containing said symbolic reference.

17. In a computer system comprising a program, a method for executing said program comprising the steps of:

receiving intermediate form object code for said program with symbolic data references in certain instructions of said intermediate form object code; and

converting the instructions of the intermediate form object code having symbolic data references, said converting step comprising the substeps of:

resolving said symbolic references to corresponding numerical references,

storing said numerical references, and

obtaining data in accordance to said numerical references.

18. A computer-implemented method for executing program operations, each operation being comprised of a set of instructions, certain of said instructions containing one or more symbolic references, said method comprising the steps of:

receiving a set of instructions reflecting an operation; and

performing the operation corresponding to the received set of instructions, wherein at least one of said symbolic references is resolved by determining a numerical reference corresponding to said symbolic reference, storing said numerical reference, and obtaining data in accordance to said stored numerical reference.

19. A memory for use in executing a program by a processor, the memory comprising:

intermediate form code containing symbolic field references associated with an intermediate representation of source code for the program,

the intermediate representation having been generated by lexically analyzing the source code and parsing output of said lexical analysis, and

wherein the symbolic field references are resolved by determining a numerical reference corresponding to said symbolic reference, and storing said numerical reference in a memory.

20. A computer-implemented method comprising:

receiving a program that comprises a set instructions written in an intermediate form code;

replacing each instruction in the program with a symbolic data reference with a new instruction containing a numeric reference resulting from invocation of a dynamic field reference routine to resolve the symbolic data reference; and

executing the program by performing an operation in accordance with each instruction or new instruction, depending upon whether an instruction has been replaced with a new instruction in accordance with the replacing step.

21. A data processing system, comprising:

a processor; and

a memory comprising a control program for causing the processor to (i) receive a program that comprises a set instructions written in an intermediate form code, (ii) replace each instruction in the program with a symbolic data reference with a new instruction containing a numeric reference resulting from invocation of a dynamic field reference routine to resolve the symbolic data reference, and (iii) execute the program by performing an operation in accordance with each instruction or new instruction, depending upon whether an instruction has been replaced with a new instruction in accordance with the replacing step.

22. An apparatus comprising:

a memory containing a compiled program in intermediate form object code constituted by a set of instructions, at least one of the instructions containing a symbolic reference; and

a processor configured to execute the instruction by determining a numerical reference corresponding to the symbolic reference, and performing an operation in accordance with the instruction and data obtained in accordance with the numerical reference without recompiling the program or any portion thereof.

23. A computer-implemented method, comprising:

receiving a program with a set instructions written in an intermediate form code;

analyzing each instruction of the program to determine whether the instruction contains a symbolic reference to a data object; and

executing the program, wherein when it was determined that an instruction contains a symbolic reference, data

from a storage location identified by a numeric reference correspoding to the symbolic reference is used thereafter to perform an operation corresponding to that instruction.

24. A computer-implemented method for executing a program comprised of bytecodes, the method comprising:

determining immediately prior to execution whether a bytecode of the program contains a symbolic data reference;

when it is determined that the bytecode of the program contains a symbolic data reference, invoking a dynamic field reference routine to resolve the symbolic data reference; and

executing thereafter the bytecode using stored data located using a numeric reference resulting from the resolution of the symbolic reference.

25. A data processing system, comprising:

a processor; and

a memory comprising a program comprised of bytecodes and instructions for causing the processor to (*i*) determine immediately prior to execution of the program whether a bytecode of the program contains a symbolic data reference, (*ii*) when it is determined that the bytecode of the program contains a symbolic data reference, invoke a dynamic field reference routine to resolve the symbolic data reference, and (*iii*) execute thereafter the bytecode using stored data located using a numeric reference resulting from the resolution of the symbolic reference.

26. A computer program product containing instructions for causing a computer to perform a method for executing a program comprised of bytecodes, the method comprising:

determining immediately prior to execution whether a bytecode of the program contains a symbolic data reference;

when it is determined that the bytecode of the program contains a symbolic data reference, invoking a dynamic field reference routine to resolve the symbolic data reference; and

executing thereafter the bytecode using stored data located using a numeric reference resulting from the resolution of the symbolic reference.

27. A computer-implemented method comprising:

receiving a program with a set of original instructions written in an intermediate form code;

generating a set of new instructions for the program that contain numeric references resulting from invocation of a routine to resolve any symbolic data references in the set of original instructions; and

executing the program using the set of new instructions.

28. A data processing system, comprising:

a processor; and

a memory comprising a control program for causing the processor to (*i*) receive a program with a set of original instructions written in an intermediate form code, (*ii*) generate a set of new instructions for the program that contain numeric references resulting from invocation of a routine to resolve any symbolic data references in the set of original instructions, and (*iii*) executing the program using the set of new instructions.

29. A computer program product containing instructions for causing a computer to perform a method, the method comprising:

receiving a program with a set of original instructions written in an intermediate form code;

generating a set of new instructions for the program that contain numeric references resulting from invocation of a routine to resolve any symbolic data references in the set of original instructions; and

executing the program using the set of new instructions.

30. A computer-implemented method comprising:

receiving a program that comprises a set instructions written in an intermediate form code;

replacing each instruction in the program with a symbolic data reference with a new instruction containing a numeric reference resulting from invocation of a dynamic field reference routine to resolve the symbolic data reference; and

executing the program by performing an operation in accordance with each instruction or new instruction, depending upon whether an instruction has been replaced with a new instruction in accordance with the replacing step.

31. A data processing system, comprising:

a processor; and

a memory comprising a control program for causing the processor to (*i*) receive a program that comprises a set instructions written in an intermediate form code, (*ii*) replace each instruction in the program with a symbolic data reference with a new instruction containing a numeric reference resulting from invocation of a dynamic field reference routing to resolve the symbolic data reference, and (*iii*) execute the program by performing an operation in accordance with each instruction or new instruction, depending upon whether an instruction has been replaced with a new instruction in accordance with the replacing step.

32. A computer program product containing control instructions for causing a computer to perform a method, the method comprising:

receiving a program that comprises a set instructions written in an intermediate form code;

replacing each instruction in the program with a symbolic data reference with a new instruction containing a numeric reference resulting from invocation of a dynamic field reference routine to resolve the symbolic data reference; and

executing the program by performing an operation in accordance with each instruction or new instruction, depending upon whether an instruction has been replaced with a new instruction in accordance with the replacing step.

33. A computer-implemented method, comprising:

receiving a program with a set instructions written in an intermediate form code;

analyzing each instruction of the program to determine whether the instruction contains a symbolic reference to a data object; and

executing the program, wherein when it was determined that an instruction contains a symbolic reference, data from a storage location identified by a numeric reference corresponding to the symbolic reference is used thereafter to perform an operation corresponding to that instruction.

34. A data processing system, comprising:

a processor; and

a memory comprising a control program for causing the processor to (*i*) receive a program with a set instructions written in an intermediate form code, (*ii*) analyze each instruction of the program to determine whether

the instruction contains a symbolic reference to a data object, and (iii) execute the program, wherein when it was determined that an instruction contains a symbolic reference, data from a storage location identified by a numeric reference corresponding to the symbolic reference is used thereafter to perform an operation corresponding to that instruction.

35. A computer program product containing control instructions for causing a computer to perform a method, the method comprising:

receiving a program with a set instructions written in an intermediate form code;

analyzing each instruction of the program to determine whether the instruction contains a symbolic reference to a data object; and

executing the program, wherein when it was determined that an instruction contains a symbolic reference, data from a storage location identified by a numeric reference corresponding to the symbolic reference is used thereafter to perform an operation corresponding to that instruction.

36. A computer-implemented method for executing a program comprised of bytecodes, the method comprising:

determining whether a bytecode of the program contains a symbolic reference;

when it is determined that the bytecode contains a symbolic reference, invoking a dynamic field reference routine to resolve the symbolic reference; and

performing an operation identified by the bytecode thereafter using data from a storage location identified by a numeric reference resulting from the invocation of the dynamic field reference routine.

37. A data processing system, comprising:

a processor; and

a memory comprising a program comprised of bytecodes and instructions for causing the processor to (i) determine whether a bytecode of the program contains a symbolic reference, (ii) when it is determined that the bytecode contains a symbolic reference, invoke a dynamic field reference routine to resolve the symbolic reference, and (iii) perform an operation identified by the bytecode thereafter using data from a storage location identified by a numeric reference resulting from the invocation of the dynamic field reference routine.

38. A computer program product containing instructions for causing a computer to perform a method for executing a program comprised of bytecodes, the method comprising:

determining whether a bytecode of the program contains a symbolic reference;

when it is determined that the bytecode contains a symbolic reference, invoking a dynamic field reference routine to resolve the symbolic reference; and

performing an operation identified by the bytecode therafter using data from a storage location identified by a numeric reference resulting from the invocation of the dynamic field reference routine.

39. A computer-implemented method comprising:

receiving a program formed of instructions written in an intermediate form code compiled from source code;

analyzing each instruction to determine whether it contains a symbolic field reference; and

executing the program by performing an operation identified by each instruction, wherein data from a storage location identified by a numeric reference is thereafter used for the operation when the instruction contains a symbolic field reference, the numeric reference having been resolved from the symbolic field reference.

40. A data processing system, comprising:

a processor; and

a memory comprising a control program for causing the processor to (i) receive a program formed of instructions written in an intermediate form code compiled from source code, (ii) analyze each instruction to determine whether it contains a symbolic field reference, and (iii) execute the program by performing an operation identified by each instruction, wherein data from a storage location identified by a numeric reference is thereafter used for the operation when the instruction contains a symbolic field reference, the numeric reference having been resolved from the symbolic field reference.

41. A computer program product containing control instructions for causing a computer to perform a method, the method comprising:

receiving a program formed of instructions written in an intermediate form code compiled from source code;

analyzing each instruction to determine whether it contains a symbolic field reference; and

executing the program by performing an operation identified by each instruction, wherein data from a storage location identified by a numeric reference is used thereafter for the operation when the instruction contains a symbolic field reference, the numeric reference having been resolved from the symbolic field reference.

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : RE 38,104 E
DATED : April 29, 2003
INVENTOR(S) : James Gosling

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

Title Page,
Item (57), **ABSTRACT**,
Line 24, please replace "interaction in" with -- instruction is --.

Column 6,
Line 15, please replace "interleaving" with -- interleavingly --.
Line 18, after "wherein", please insert a comma.
Line 23, after "counter", please insert a semicolon.
Line 25, please replace "said c)" with -- said step c) --.

Column 7,
Lines 3-4, please replace "and program execution" with -- said program execution --.
Lines 38, 41 and 45, please replace " 3" with -- 13 --.

Column 8,
Lines 20-33, delete the claim in its entirety and insert therefore:

--20. A computer-implemented method for executing a compiled program containing instructions in an intermediate form code, at least one of the instructions containing a symbolic reference, said method comprising the steps of:

resolving the symbolic reference in the instruction by determining a numerical reference corresponding to the symbolic reference; and

performing an operation in accordance with the instruction and data obtained in accordance with the numerical reference without recompiling the program or any portion thereof.--.

Lines 34-47, delete the claim in its entirety and insert therefore:

--21. A memory encoded with a compiled program, the memory comprising:

intermediate form code containing symbolic field references associated with an intermediate representation of source code for the program,

the intermediate representation having been generated by lexically analyzing the source code and parsing output of said lexical analysis,

such that when the program is executed by a processor each symbolic field reference is resolved by determining a numerical reference corresponding to the symbolic field reference and data is obtained in accordance with the numerical reference without recompiling the program or any portion thereof.--.

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : RE 38,104 E
DATED : April 29, 2003
INVENTOR(S) : James Gosling

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

Column 8, line 60 - Column 9, line 4,
Delete the claim in its entirety and insert therefore:

-- 23. A computer-readable medium containing instructions for controlling a data processing system to perform a method for interpreting a compiled program in intermediate form object code comprised of instructions, at least one of the instructions containing a symbolic reference, said method comprising the steps of:

resolving the symbolic reference in the instruction by determining a numerical reference corresponding to the symbolic reference; and

performing an operation in accordance with the instruction and data obtained in accordance with the numerical reference without recompiling the program or any portion thereof. --.

Column 10,
Lines 7, 23-24, 37, 50 and 65-66, please replace "*a set instructions*" with -- *a set of instructions* --.
Line 28, please replace "*routing*" with -- *routine* --.

Column 11,
Line 11, please replace "*a set instructions*" with -- *a set of instructions* --.

Column 12,
Line 7, please replace "*therafter*" with -- *thereafter* --.

Signed and Sealed this

Sixteenth Day of September, 2003

JAMES E. ROGAN
Director of the United States Patent and Trademark Office

EXHIBIT F

US006910205B2

(12) **United States Patent**
Bak et al.

(10) **Patent No.: US 6,910,205 B2**
(45) **Date of Patent: *Jun. 21, 2005**

(54) **INTERPRETING FUNCTIONS UTILIZING A HYBRID OF VIRTUAL AND NATIVE MACHINE INSTRUCTIONS**

(75) Inventors: **Lars Bak**, Palo Alto, CA (US); **Robert Griesemer**, Menlo Park, CA (US)

(73) Assignee: **Sun Microsystems, Inc.**, Santa Clara, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 175 days.

This patent is subject to a terminal disclaimer.

(21) Appl. No.: **10/194,040**

(22) Filed: **Jul. 12, 2002**

(65) **Prior Publication Data**

US 2002/0184399 A1 Dec. 5, 2002

Related U.S. Application Data

(63) Continuation of application No. 08/884,856, filed on Jun. 30, 1997, now Pat. No. 6,513,156.

(51) **Int. Cl.**[7] **G06F 9/45**; G06F 9/455

(52) **U.S. Cl.** **717/151**; 717/159; 717/148; 717/139; 718/1

(58) **Field of Search** 717/151, 159, 717/139, 148; 718/1

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,329,611	A		7/1994	Pechanek et al.
5,367,685	A		11/1994	Gosling
5,586,328	A		12/1996	Caron et al.
5,758,162	A		5/1998	Takayama et al.
5,768,593	A		6/1998	Walters et al.
5,845,298	A		12/1998	O'Conner et al.
5,898,850	A		4/1999	Dickol et al.
5,905,895	A		5/1999	Halter
5,925,123	A		7/1999	Tremblay et al.
5,953,736	A		9/1999	O'Conner et al.
5,995,754	A	*	11/1999	Holzle et al. 717/158
6,038,394	A		3/2000	Layes

(Continued)

OTHER PUBLICATIONS

Proebsting, Todd A., "Optimizing an ANSI C Interpreter with Superoperators," pp. 322–332, Jan. 1995.

Hsieh, Cheng–Hsueh et al., "Java Bytecode to Native Code Translation: The caffeine prototype and preliminary results," pp. 90–97, Dec. 1996.

Lambright, H. Dan., "Java Bytecode Optimizations," pp. 206–210, Feb. 1997.

Pittan Thomas, "Two–level Hybrid Interpreter/Native Code Execution for combined space time program efficiency," ACM, pp. 150–152, Jun. 1987.

Kaufer, Stephen et al., "Saber–C, An Interpreter–based programming environment for the C language," USENIX, pp. 161–171, Jun. 1988.

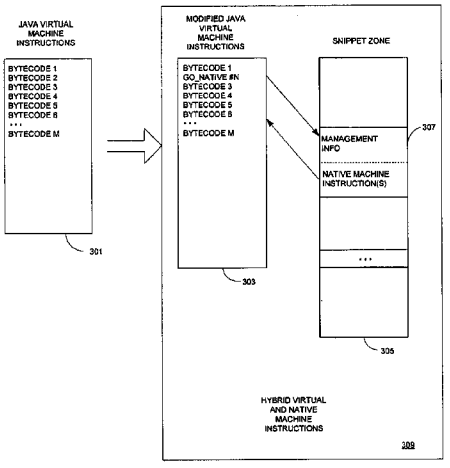
Davidson, Jack W. et al., "Cint: A RISC Interpreter for the C programming language," ACM, pp. 189–198, Jun. 1987.

Primary Examiner—Lewis A. Bullock, Jr.
(74) *Attorney, Agent, or Firm*—Beyer Weaver & Thomas LLP

(57) **ABSTRACT**

Systems and methods for increasing the execution speed of virtual machine instructions for a function are provided. A portion of the virtual machine instructions of the function are compiled into native machine instructions so that the function includes both virtual and native machine instructions. Execution of the native machine instructions may be accomplished by overwriting a virtual machine instruction of the function with a virtual machine instruction that specifies execution of the native machine instructions. Additionally, the original virtual machine instruction may be stored so that the original virtual machine instructions can be regenerated.

14 Claims, 12 Drawing Sheets



U.S. PATENT DOCUMENTS

6,044,220	A	3/2000	Breternitz	
6,118,940	A	9/2000	Alexander et al.	
6,170,083	B1	1/2001	Adl-Tabatabai	
6,292,883	B1 *	9/2001	Augusteijn et al.	712/209
6,332,216	B1	12/2001	Manjunath	
6,349,377	B1 *	2/2002	Lindwer	712/22

* cited by examiner

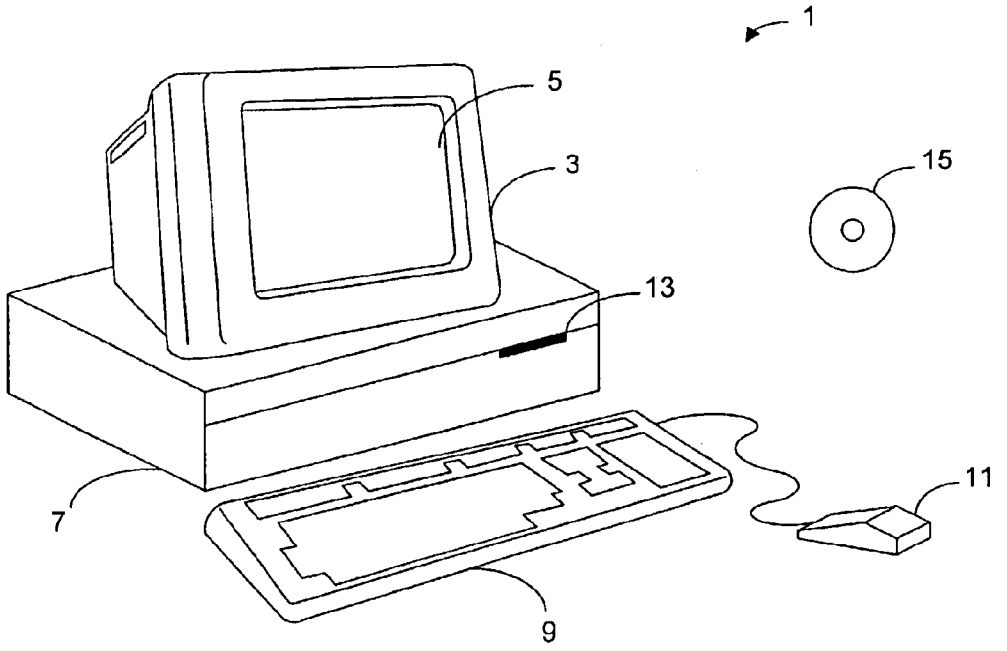

1
5
3
15
13
11
7
9

FIG. 1

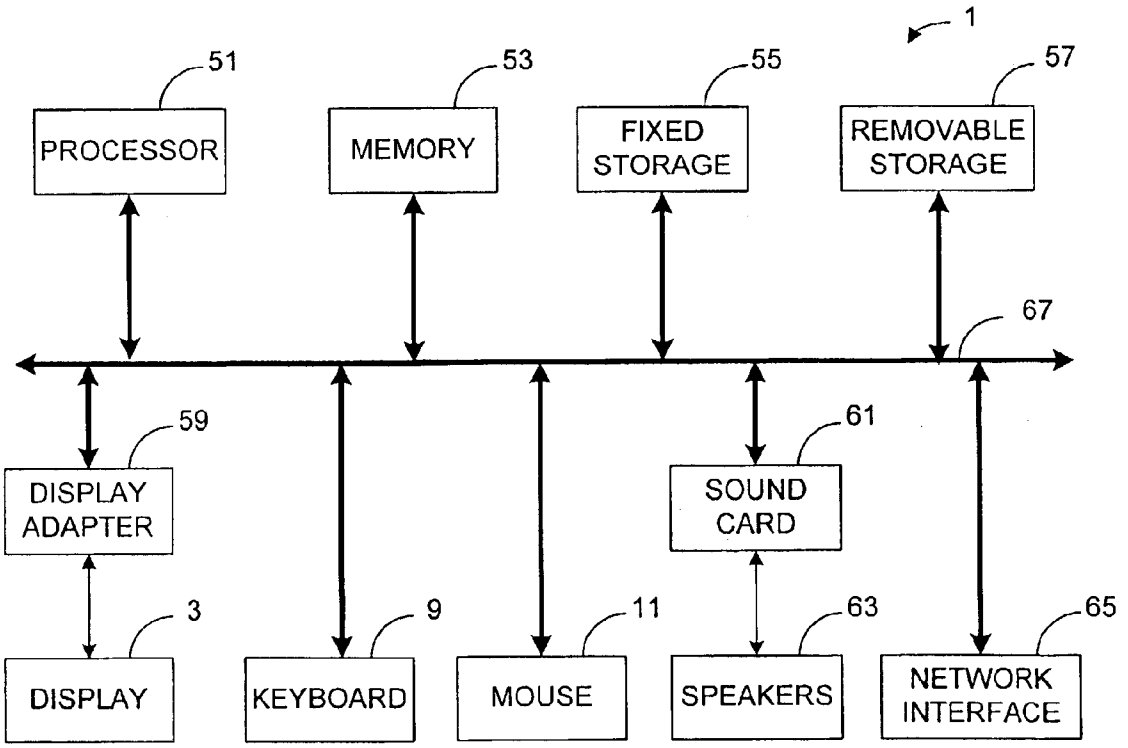

1
51
PROCESSOR
53
MEMORY
55
FIXED STORAGE
57
REMOVABLE STORAGE
67
59
DISPLAY ADAPTER
61
SOUND CARD
3
DISPLAY
9
KEYBOARD
11
MOUSE
63
SPEAKERS
65
NETWORK INTERFACE

FIG. 2

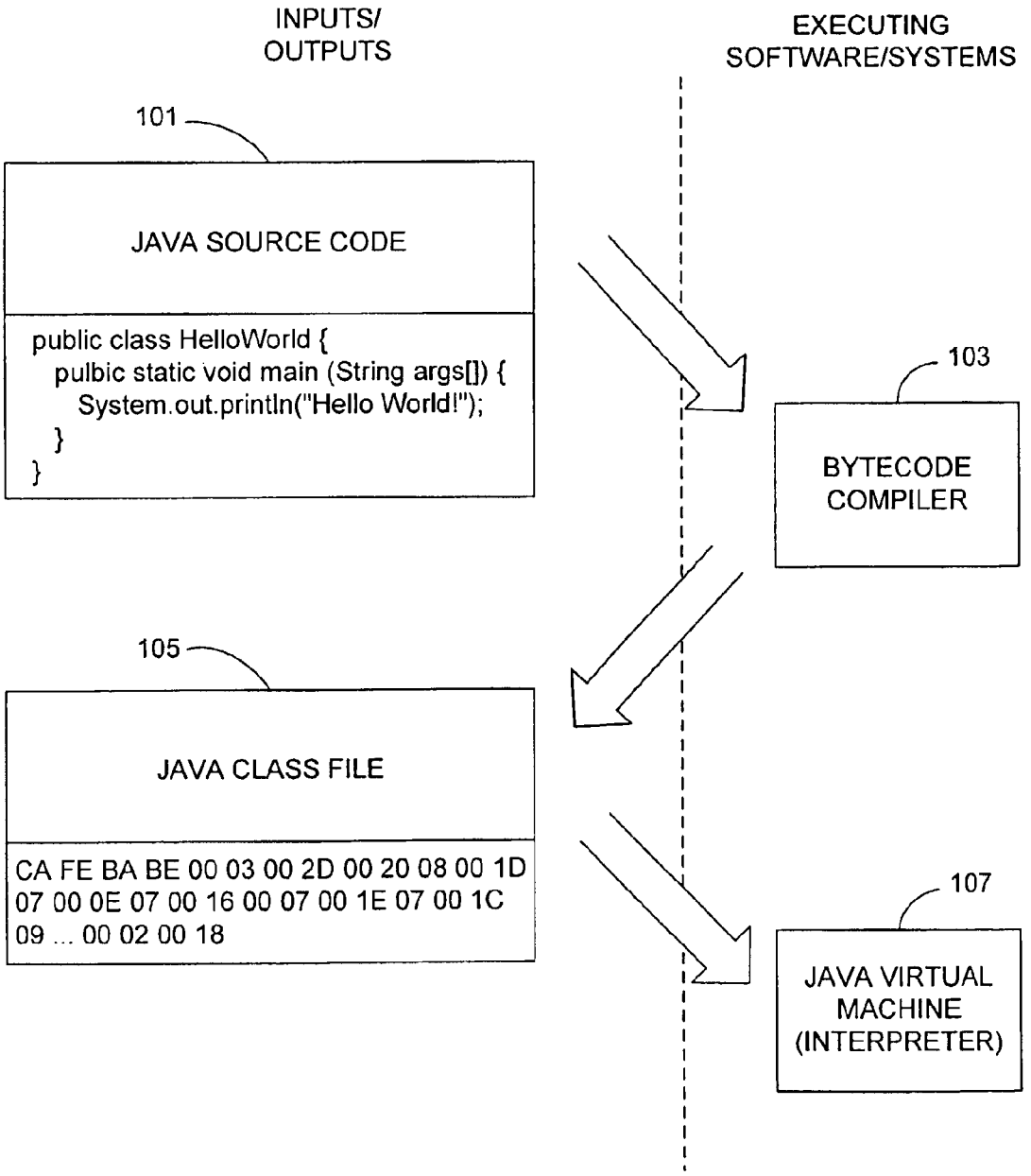

INPUTS/
OUTPUTS
EXECUTING
SOFTWARE/SYSTEMS
101
JAVA SOURCE CODE
public class HelloWorld {
pulbic static void main (String args[]) {
System.out.println("Hello World!");
}
}
103
BYTECODE
COMPILER
105
JAVA CLASS FILE
CA FE BA BE 00 03 00 2D 00 20 08 00 1D
07 00 0E 07 00 16 00 07 00 1E 07 00 1C
09 ... 00 02 00 18
107
JAVA VIRTUAL
MACHINE
(INTERPRETER)

FIG. 3

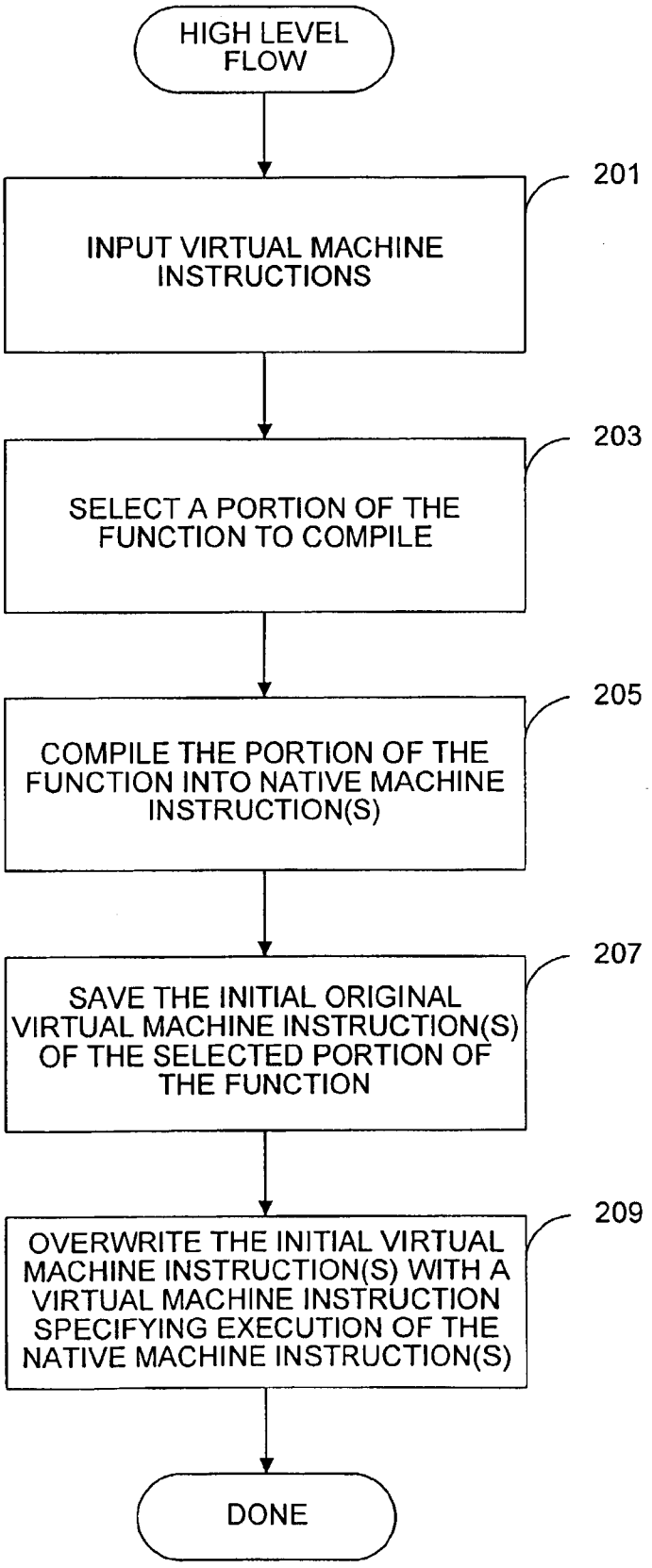

HIGH LEVEL FLOW
201
INPUT VIRTUAL MACHINE INSTRUCTIONS
203
SELECT A PORTION OF THE FUNCTION TO COMPILE
205
COMPILE THE PORTION OF THE FUNCTION INTO NATIVE MACHINE INSTRUCTION(S)
207
SAVE THE INITIAL ORIGINAL VIRTUAL MACHINE INSTRUCTION(S) OF THE SELECTED PORTION OF THE FUNCTION
209
OVERWRITE THE INITIAL VIRTUAL MACHINE INSTRUCTION(S) WITH A VIRTUAL MACHINE INSTRUCTION SPECIFYING EXECUTION OF THE NATIVE MACHINE INSTRUCTION(S)
DONE

FIG. 4

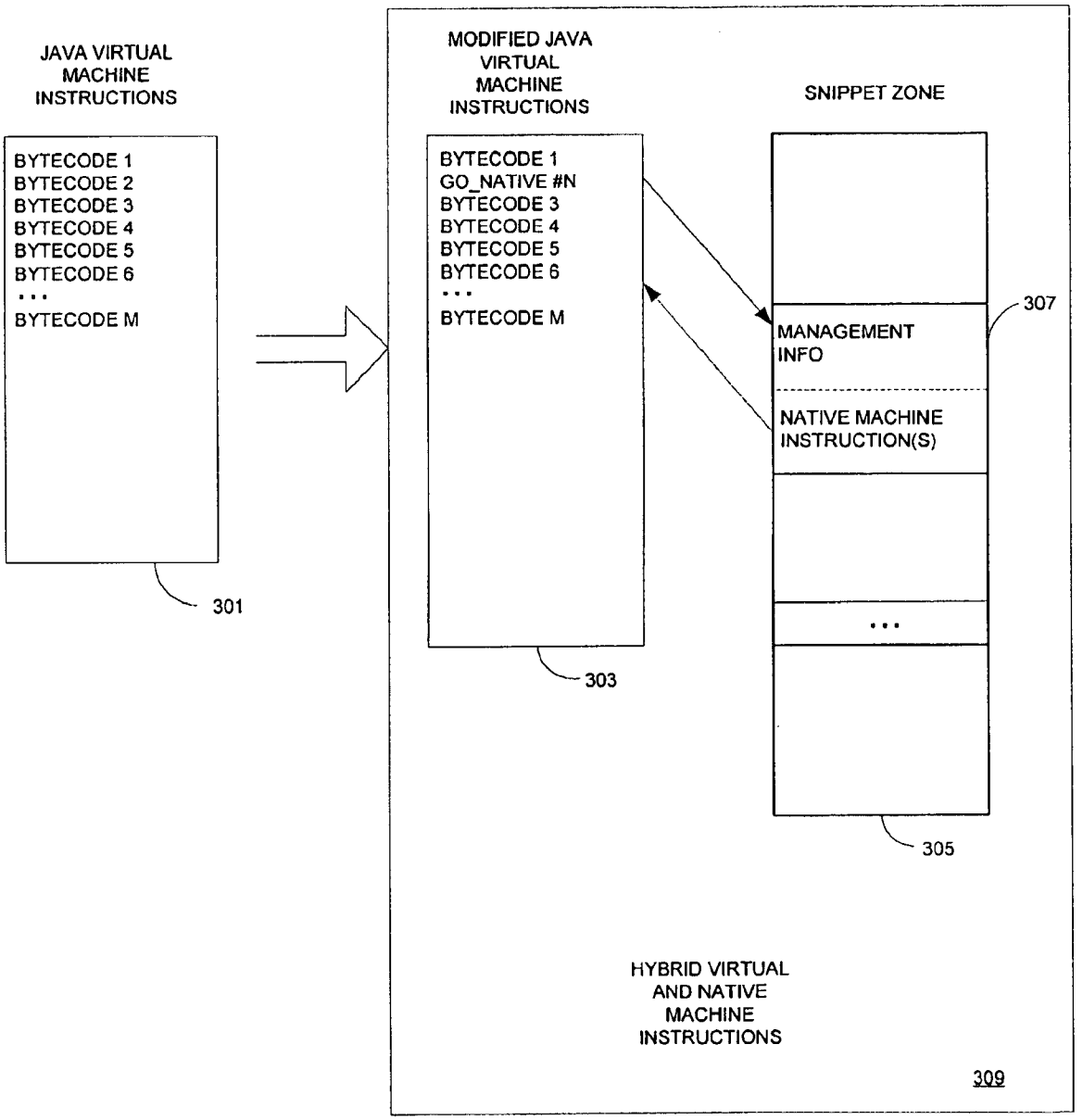

JAVA VIRTUAL MACHINE INSTRUCTIONS
BYTECODE 1
BYTECODE 2
BYTECODE 3
BYTECODE 4
BYTECODE 5
BYTECODE 6
...
BYTECODE M
301
MODIFIED JAVA VIRTUAL MACHINE INSTRUCTIONS
BYTECODE 1
GO_NATIVE #N
BYTECODE 3
BYTECODE 4
BYTECODE 5
BYTECODE 6
...
BYTECODE M
303
SNIPPET ZONE
MANAGEMENT INFO
307
NATIVE MACHINE INSTRUCTION(S)
...
305
HYBRID VIRTUAL AND NATIVE MACHINE INSTRUCTIONS
309

FIG. 5

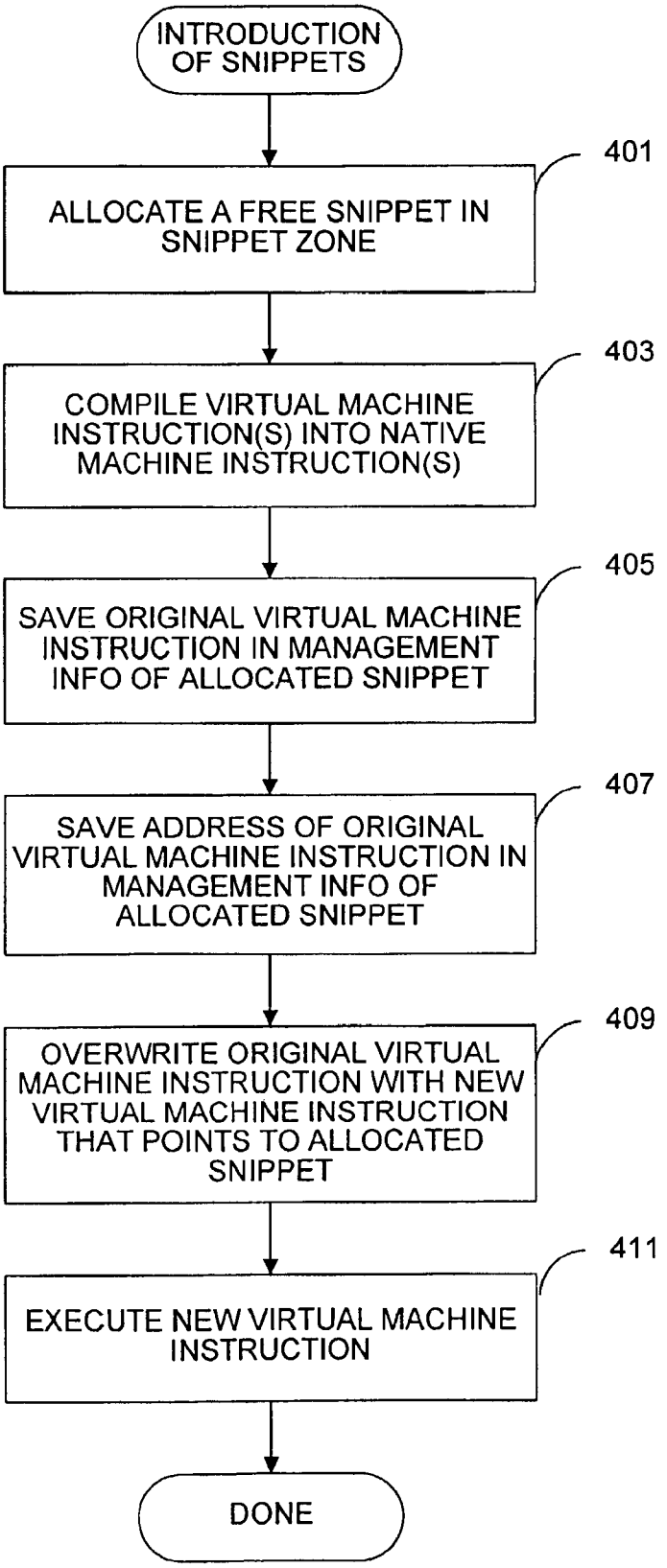

INTRODUCTION OF SNIPPETS
401
ALLOCATE A FREE SNIPPET IN SNIPPET ZONE
403
COMPILE VIRTUAL MACHINE INSTRUCTION(S) INTO NATIVE MACHINE INSTRUCTION(S)
405
SAVE ORIGINAL VIRTUAL MACHINE INSTRUCTION IN MANAGEMENT INFO OF ALLOCATED SNIPPET
407
SAVE ADDRESS OF ORIGINAL VIRTUAL MACHINE INSTRUCTION IN MANAGEMENT INFO OF ALLOCATED SNIPPET
409
OVERWRITE ORIGINAL VIRTUAL MACHINE INSTRUCTION WITH NEW VIRTUAL MACHINE INSTRUCTION THAT POINTS TO ALLOCATED SNIPPET
411
EXECUTE NEW VIRTUAL MACHINE INSTRUCTION
DONE

FIG. 6

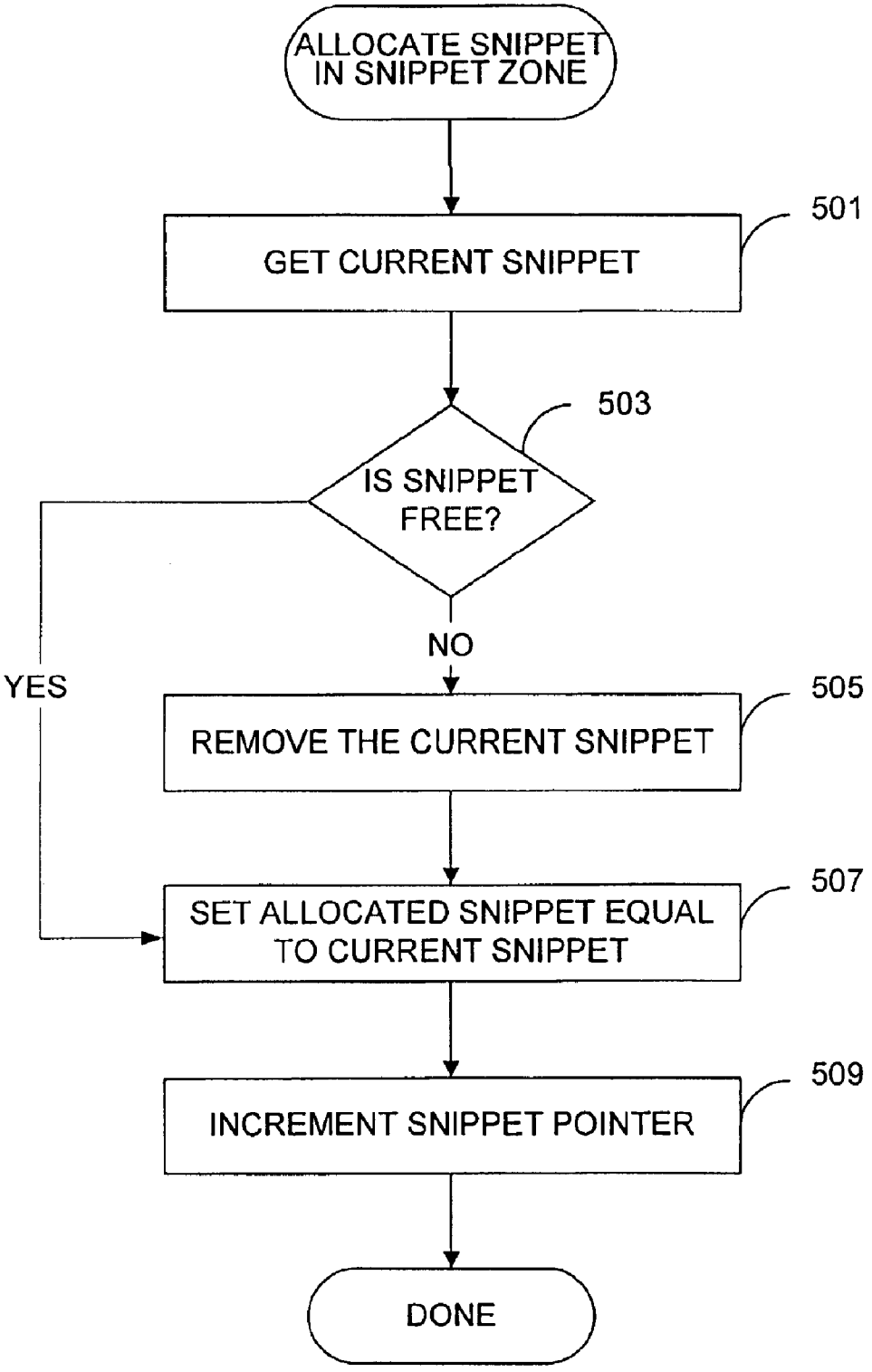

ALLOCATE SNIPPET IN SNIPPET ZONE
GET CURRENT SNIPPET
501
503
IS SNIPPET FREE?
YES
NO
REMOVE THE CURRENT SNIPPET
505
SET ALLOCATED SNIPPET EQUAL TO CURRENT SNIPPET
507
INCREMENT SNIPPET POINTER
509
DONE

FIG. 7

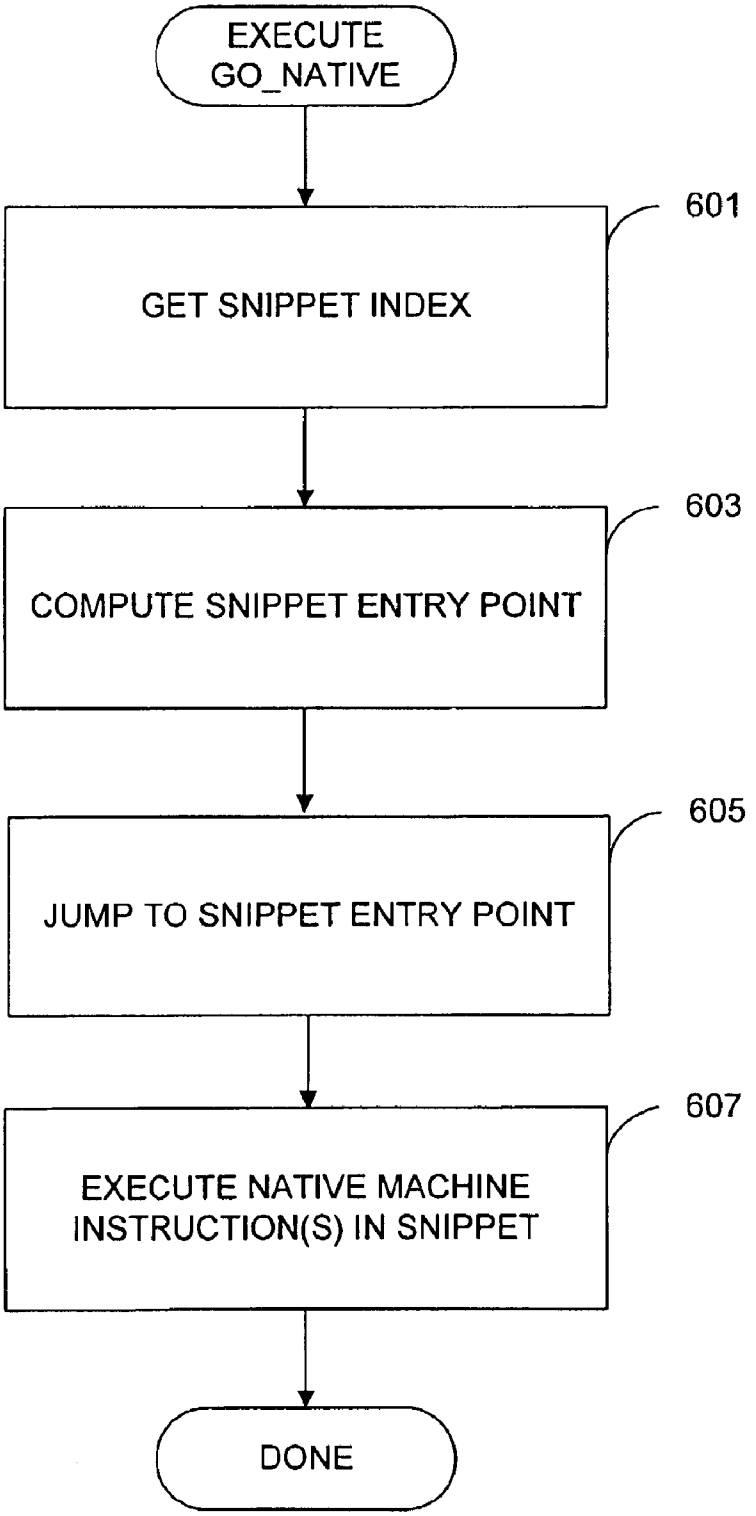

EXECUTE GO_NATIVE
601
GET SNIPPET INDEX
603
COMPUTE SNIPPET ENTRY POINT
605
JUMP TO SNIPPET ENTRY POINT
607
EXECUTE NATIVE MACHINE INSTRUCTION(S) IN SNIPPET
DONE

FIG. 8

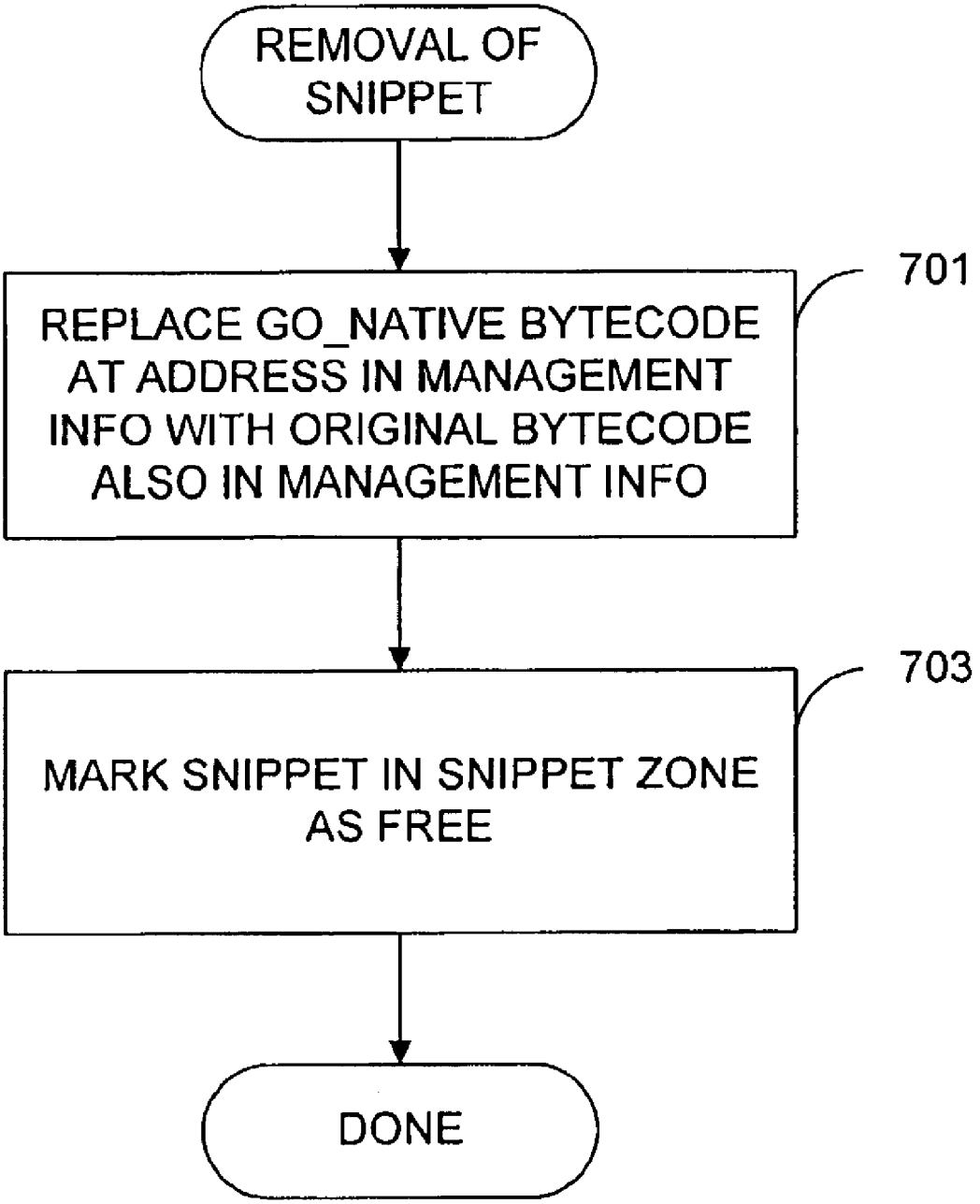

REMOVAL OF SNIPPET
701
REPLACE GO_NATIVE BYTECODE AT ADDRESS IN MANAGEMENT INFO WITH ORIGINAL BYTECODE ALSO IN MANAGEMENT INFO
703
MARK SNIPPET IN SNIPPET ZONE AS FREE
DONE

FIG. 9

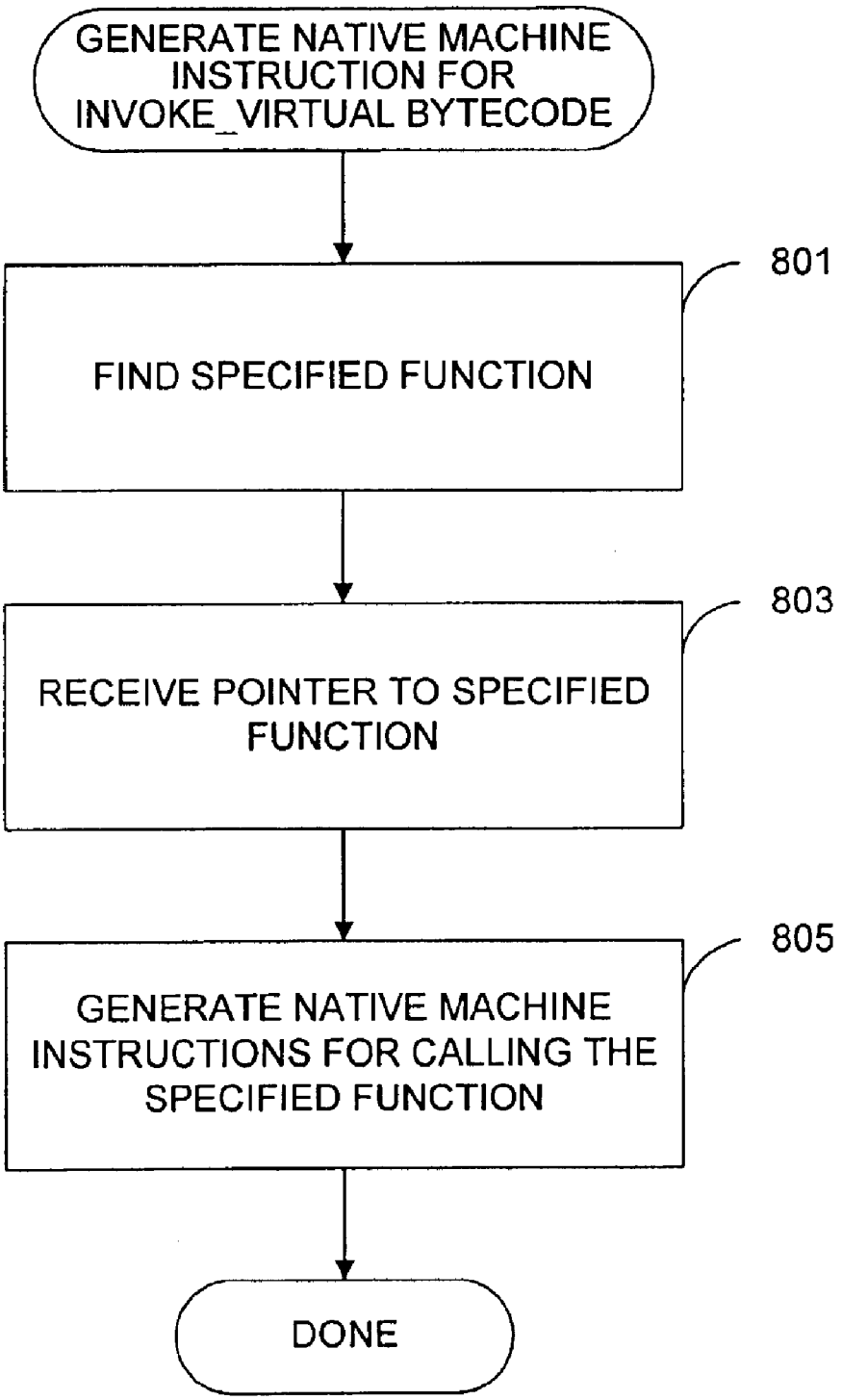

GENERATE NATIVE MACHINE INSTRUCTION FOR INVOKE_VIRTUAL BYTECODE
801
FIND SPECIFIED FUNCTION
803
RECEIVE POINTER TO SPECIFIED FUNCTION
805
GENERATE NATIVE MACHINE INSTRUCTIONS FOR CALLING THE SPECIFIED FUNCTION
DONE

FIG. 10

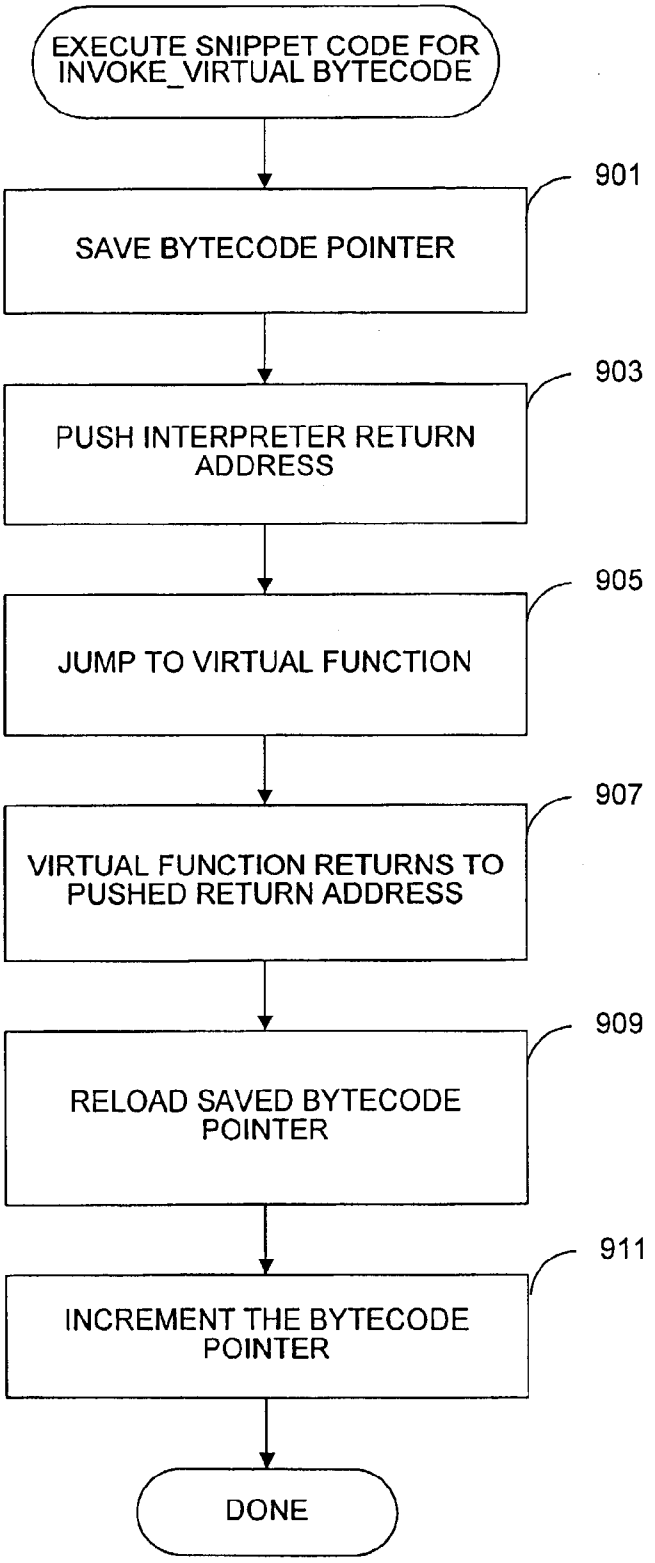

EXECUTE SNIPPET CODE FOR INVOKE_VIRTUAL BYTECODE
901
SAVE BYTECODE POINTER
903
PUSH INTERPRETER RETURN ADDRESS
905
JUMP TO VIRTUAL FUNCTION
907
VIRTUAL FUNCTION RETURNS TO PUSHED RETURN ADDRESS
909
RELOAD SAVED BYTECODE POINTER
911
INCREMENT THE BYTECODE POINTER
DONE

FIG. 11

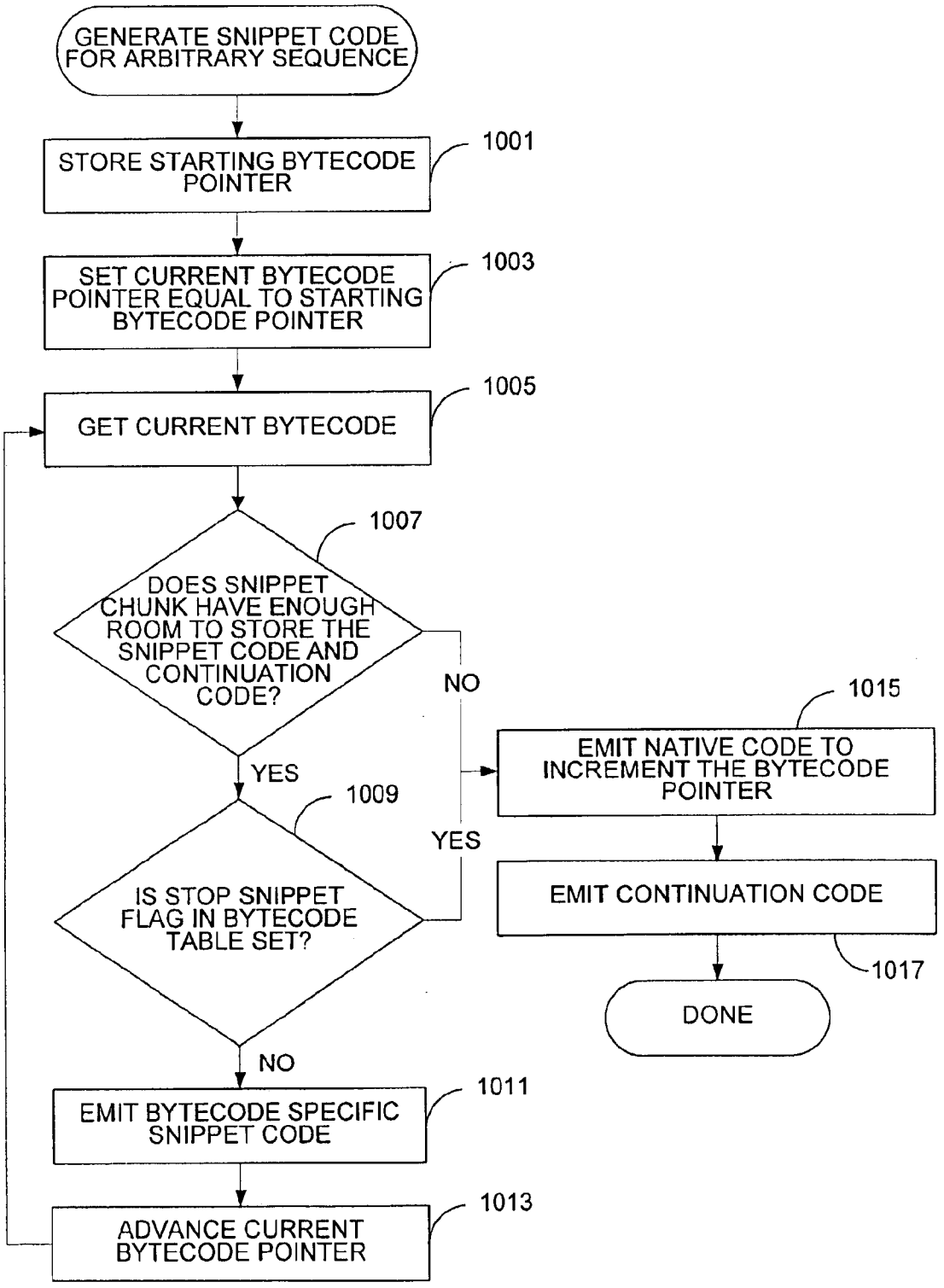

GENERATE SNIPPET CODE FOR ARBITRARY SEQUENCE
STORE STARTING BYTECODE POINTER
1001
SET CURRENT BYTECODE POINTER EQUAL TO STARTING BYTECODE POINTER
1003
GET CURRENT BYTECODE
1005
DOES SNIPPET CHUNK HAVE ENOUGH ROOM TO STORE THE SNIPPET CODE AND CONTINUATION CODE?
1007
NO
YES
IS STOP SNIPPET FLAG IN BYTECODE TABLE SET?
1009
YES
NO
EMIT NATIVE CODE TO INCREMENT THE BYTECODE POINTER
1015
EMIT CONTINUATION CODE
1017
DONE
EMIT BYTECODE SPECIFIC SNIPPET CODE
1011
ADVANCE CURRENT BYTECODE POINTER
1013

FIG. 12

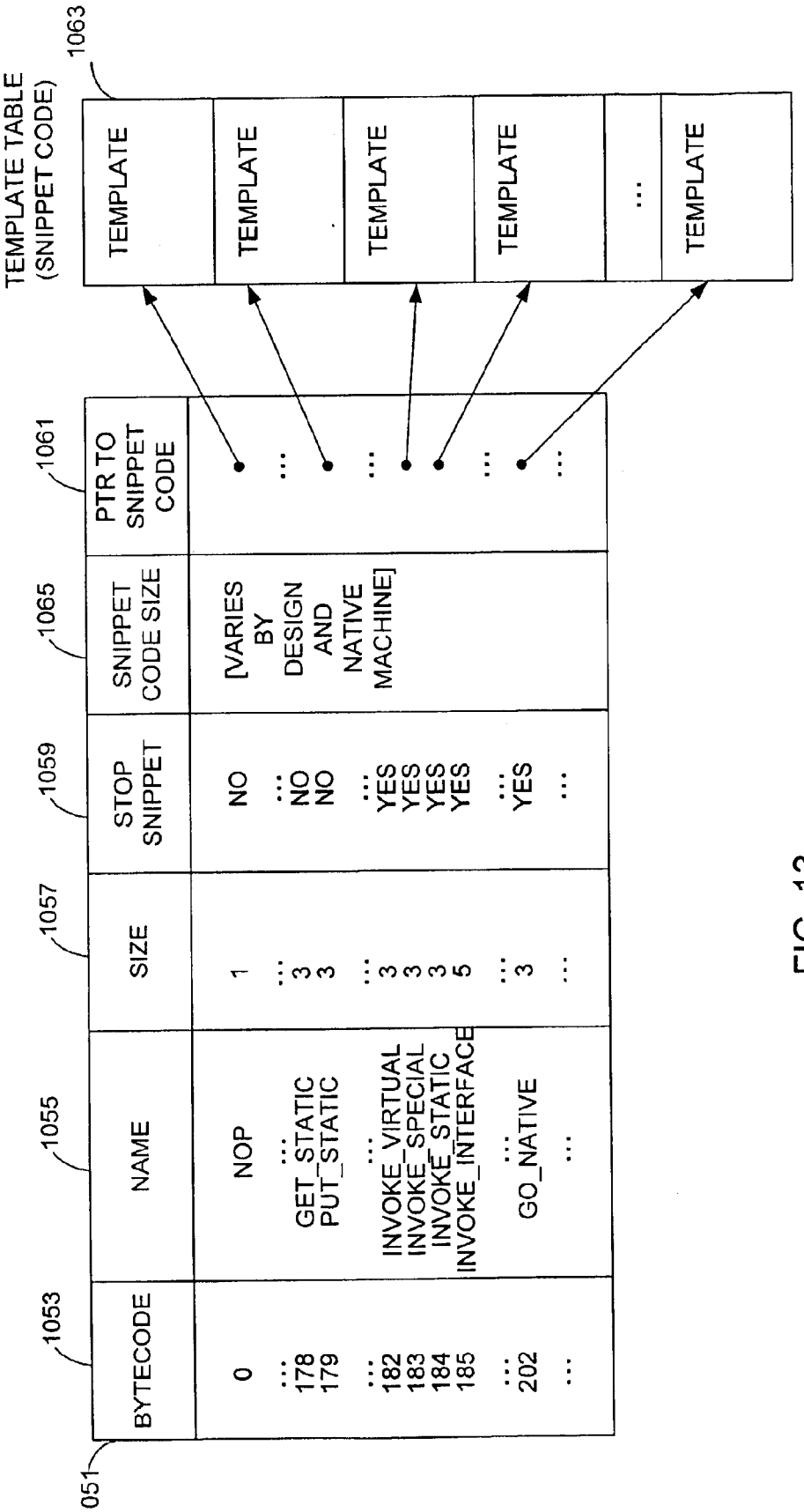

TEMPLATE TABLE
(SNIPPET CODE)
1063
TEMPLATE
TEMPLATE
TEMPLATE
TEMPLATE
...
TEMPLATE
1061
PTR TO SNIPPET CODE
1065
SNIPPET CODE SIZE
[VARIES BY DESIGN AND NATIVE MACHINE]
1059
STOP SNIPPET
NO
NO
NO
YES
YES
YES
YES
YES
1057
SIZE
1
3
3
3
3
3
5
3
1055
NAME
NOP
GET_STATIC
PUT_STATIC
INVOKE_VIRTUAL
INVOKE_SPECIAL
INVOKE_STATIC
INVOKE_INTERFACE
GO_NATIVE
1053
BYTECODE
0
178
179
182
183
184
185
202
051

FIG. 13

INTERPRETING FUNCTIONS UTILIZING A HYBRID OF VIRTUAL AND NATIVE MACHINE INSTRUCTIONS

This is a Continuation application of prior application Ser. No. 08/884,856 filed on Jun. 30, 1997, now U.S. Pat. No. 6,513,156 the disclosure of which is incorporated herein by reference.

BACKGROUND OF THE INVENTION

The present invention relates to increasing the execution speed of interpreters and, more specifically, increasing the speed of an interpreter executing a Java™ function utilizing a hybrid of virtual and native machine instructions.

The computer era began back in the early 1950s with the development of the UNIVAC. Today, there are countless numbers of computers and computer platforms. Although the variety of computers is a blessing for users, it is a curse for programmers and application developers who have the unfortunate task of modifying or porting an existing computer program to run on a different computer platform.

One of the goals of high level languages is to provide a portable programming environment such that the computer programs may be easily ported to another computer platform. High level languages such as "C" provide a level of abstraction from the underlying computer architecture and their success is well evidenced from the fact that most computer applications are now written in a high level language.

Portability has been taken to new heights with the advent of World Wide Web ("the Web") which is an interface protocol for the Internet which allows communication of diverse computer platforms through a graphical interface. Computers communicating over the Web are able to download and execute small applications called applets. Given that applets may be executed on a diverse assortment of computer platforms, the applets are typically executed by a Java™ virtual machine.

The Java™ programming language is an object-oriented high level programming language developed by Sun Microsystems and designed to be portable enough to be executed on a wide range of computers ranging from small personal computers up to supercomputers. Java programs are compiled into class files which include virtual machine instructions (e.g., bytecodes) for the Java virtual machine. The Java virtual machine is a software emulator of a "generic" computer. An advantage of utilizing virtual machine instructions is the flexibility that is achieved since the virtual machine instructions may be run, unmodified, on any computer system that has a virtual machine implementation, making for a truly portable language. Additionally, other programming languages may be compiled into Java virtual machine instructions and executed by a Java virtual machine.

The Java virtual machine is an interpreter executed as an interpreter program (i.e., software). Conventional interpreters decode and execute the virtual machine instructions of an interpreted program one instruction at a time during execution. Compilers, on the other hand, decode source code into native machine instructions prior to execution so that decoding is not performed during execution. Because conventional interpreters decode each instruction before it is executed repeatedly each time the instruction is encountered, execution of interpreted programs is typically quite slower than compiled programs because the native machine instructions of compiled programs can be executed on the native machine or computer system without necessitating decoding.

A known method for increasing the execution speed of Java interpreted programs of virtual machine instructions involves utilizing a just-in-tine (JIT) compiler. The JIT compiler compiles an entire Java function just before it is called. However, native code generated by a JIT compiler does not always run faster than code executed by an interpreter. For example, if the interpreter is not spending the majority of its time decoding the Java virtual machine instructions, then compiling the instructions with a JIT compiler may not increase the execution speed. In fact, execution may even be slower utilizing the JIT compiler if the overhead of compiling the instructions is more than the overhead of simply interpreting the instructions.

Another known method for increasing the execution speed of Java interpreted programs of virtual machine instructions utilizes "quick" instructions or bytecodes. The "quick" instructions take advantage of the unassigned bytecodes for the Java virtual machine. A "quick" instruction utilizes an unassigned bytecode to shadow another bytecode. The first time that the shadowed bytecode is encountered, the bytecode is replaced by the "quick" bytecode which is a more efficient implementation of the same operation. Although "quick" instructions have been implemented with good results, their flexibility is limited since the number of unassigned bytecodes is limited (and may decrease as new bytecodes are assigned).

Accordingly, there is a need for new techniques for increasing the execution speed of computer programs that are being interpreted. Additionally, there is a need for new techniques that provide flexibility in the way in which interpreted computer programs are executed.

SUMMARY OF THE INVENTION

In general, embodiments of the present invention provide innovative systems and methods for increasing the execution speed of computer programs executed by an interpreter. A portion of a function is compiled into at least one native machine instruction so that the function includes both virtual and native machine instructions during execution. With the invention, the mechanism for increasing the execution speed of the virtual machine instructions is transparent to the user, the hybrid virtual and native machine instructions may be easily transformed back to the original virtual machine instructions, and the flexibility of compiling only certain portions of a function into native machine instructions allows for better optimization of the execution of the function. Several embodiments of the invention are described below.

In one embodiment, a computer implemented method for increasing the execution speed of virtual machine instructions is provided. Virtual machine instructions for a function are input into a computer system. A portion of the function is compiled into native machine instruction(s) so that the function includes both virtual and native machine instructions. A virtual machine instruction of the function may be overwritten with a new virtual machine instruction that specifies the execution of native machine instructions that were compiled from a sequence of virtual machine instructions beginning with the overwritten virtual machine instruction of the function. In preferred embodiments, the virtual machine instructions are Java virtual machine instructions.

In another embodiment, a computer implemented method for increasing the execution speed of virtual machine instructions is provided. Java virtual machine instructions for a function are input into a computer system. A portion of the function is compiled into native machine instruction(s).

A copy of a selected virtual machine instruction at a beginning of the portion of the function is stored and a back pointer to a location of the selected virtual machine instruction is also stored. The selected virtual machine instruction is overwritten with a new virtual machine instruction that specifies execution of the native machine instructions so that the function includes both virtual and native machine instructions. The new virtual machine instruction may include a pointer to a data block in which is stored the native machine instructions, the copy of the selected virtual machine instruction, and the back pointer. Additionally, the original virtual machine instructions that were input may be generated by storing the copy of the selected virtual machine instruction stored in the data block at the location specified by the back pointer.

In another embodiment, a computer implemented method of generating hybrid virtual and native machine instructions is provided. A sequence of virtual machine instructions for a function is input into a computer system. A virtual machine instruction of the sequence of virtual machine instructions is selected and the selected virtual machine instruction is overwritten with a new virtual machine instruction that specifies one or more native machine instructions. The new virtual machine instruction may include a pointer to the one or more native machine instructions which may be stored in a data block. The one or more native machine instructions may be generated from a compilation of a portion of the sequence of virtual machine instructions beginning with the selected virtual machine instruction.

Other features and advantages of the invention will become readily apparent upon review of the following detailed description in association with the accompanying drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. **1** illustrates an example of a computer system that may be utilized to execute the software of an embodiment of the invention.

FIG. **2** shows a system block diagram of the computer system of FIG. **1**.

FIG. **3** shows how a Java source code program is executed.

FIG. **4** shows a high level flowchart illustrating a process of transforming a function into a hybrid of virtual and native machine instructions in accordance with one embodiment of the present invention.

FIG. **5** illustrates a transformation of Java virtual machine instructions of a function to hybrid virtual and native machine instructions.

FIG. **6** shows a process of introducing snippets which are native machine instructions compiled from a sequence of virtual machine instructions of a function.

FIG. **7** shows a process of allocating a snippet in the snippet zone which stores all existing snippets.

FIG. **8** shows a process of executing a go_native virtual machine instruction that specifies the execution of native machine instructions in a snippet.

FIG. **9** shows a process of removing a snippet from the hybrid, virtual, and native machine instructions of a function.

FIG. **10** shows a process of generating native machine instructions for the invoke_virtual bytecode.

FIG. **11** shows a process of executing snippet code for the invoke_virtual bytecode.

FIG. **12** shows a process of generating snippet code for an arbitrary sequence of virtual machine instructions in a function.

FIG. **13** illustrates a bytecode table which may be utilized to store information regarding different Java bytecodes.

DETAILED DESCRIPTION OF PREFERRED EMBODIMENTS

Definitions

Machine instruction—An instruction that directs a computer to perform an operation specified by an operation code (OP code) and optionally one or more operand.

Virtual machine instruction—A machine instruction for a software emulated microprocessor or computer architecture (also called virtual code).

Native machine instruction—A machine instruction that is designed for a specific microprocessor or computer architecture (also called native code).

Class—An object-oriented data type that defines the data and methods that each object of a class will include.

Function—A software routine (also called a subroutine, procedure, member and method).

Snippet—A relatively small piece of compiled native machine instructions and associated information.

Bytecode pointer (BCP)—A pointer that points to the current Java virtual machine instruction (e.g., bytecode) that is being executed.

Program counter (PC)—A pointer that points to the machine instruction of the interpreter that is being executed.

Overview

In the description that follows, the present invention will be described in reference to a preferred embodiment that increases the execution speed of Java virtual machine instructions. However, the invention is not limited to any particular language, computer architecture, or specific implementation. As an example, the invention may be advantageously applied to languages other than Java (e.g., Smalltalk). Therefore, the description of the embodiments that follow is for purposes of illustration and not limitation.

FIG. **1** illustrates an example of a computer system that may be used to execute the software of an embodiment of the invention. FIG. **1** shows a computer system **1** which includes a display **3**, screen **5**, cabinet **7**, keyboard **9**, and mouse **11**. Mouse **11** may have one or more buttons for interacting with a graphical user interface. Cabinet **7** houses a CD-ROM drive **13**, system memory and a hard drive (see FIG. **2**) which may be utilized to store and retrieve software programs incorporating computer code that implements the invention, data for use with the invention, and the like. Although the CD-ROM **15** is shown as an exemplary computer readable storage medium, other computer readable storage media including floppy disk, tape, flash memory, system memory, and hard drive may be utilized. Additionally, a data signal embodied in a carrier wave (e.g., in a network including the Internet) may be the computer readable storage medium.

FIG. **2** shows a system block diagram of computer system **1** used to execute the software of an embodiment of the invention. As in FIG. **1**, computer system **1** includes monitor **3** and keyboard **9**, and mouse **11**. Computer system **1** further includes subsystems such as a central processor **51**, system memory **53**, fixed storage **55** (e.g., hard drive), removable storage **57** (e.g., CD-ROM drive), display adapter **59**, sound card **61**, speakers **63**, and network interface **65**. Other computer systems suitable for use with the invention may include additional or fewer subsystems. For example, another computer system could include more than one processor **51** (i.e., a multi-processor system), or a cache memory.

The system bus architecture of computer system **1** is represented by arrows **67**. However, these arrows are illustrative of any interconnection scheme serving to link the subsystems. For example, a local bus could be utilized to connect the central processor to the system memory and display adapter. Computer system **1** shown in FIG. **2** is but an example of a computer system suitable for use with the invention. Other computer architectures having different configurations of subsystems may also be utilized.

Typically, computer programs written in the Java programming language are compiled into bytecodes or Java virtual machine instructions which are then executed by a Java virtual machine. The bytecodes are stored in class files which are input into the Java virtual machine for interpretation. FIG. **3** shows a progression of a simple piece of Java source code through execution by an interpreter, the Java virtual machine.

Java source code **101** includes the classic Hello World program written in Java. The source code is then input into a bytecode compiler **103** which compiles the source code into bytecodes. The bytecodes are virtual machine instructions as they will be executed by a software emulated computer. Typically, virtual machine instructions are generic (i.e., not designed for any specific microprocessor or computer architecture) but this is not required. The bytecode compiler outputs a Java class file **105** which includes the bytecodes for the Java program.

The Java class file is input into a Java virtual machine **107**. The Java virtual machine is an interpreter that decodes and executes the bytecodes in the Java class file. The Java virtual machine is an interpreter, but is commonly referred to as a virtual machine as it emulates a microprocessor or computer architecture in software (e.g., the microprocessor or computer architecture that may not exist).

An interpreter may execute a bytecode program by repeatedly executing the following steps:

Execute—execute operation of the current bytecode

Advance—advance bytecode pointer to next bytecode

Dispatch—fetch the bytecode at the bytecode pointer and jump to the implementation (i.e., execute step) of that bytecode.

The execute step implements the operation of a particular bytecode. The advance step increments the bytecode pointer so that it points to the next bytecode. Lastly, the dispatch step fetches the bytecode at the current bytecode pointer and jumps to the piece of native machine code that implements that bytecode. The execution of the execute-advance-dispatch sequence for a bytecode is commonly called an "interpretation cycle."

Although in a preferred embodiment, the interpreter utilizes the interpretation cycle described above. Many other interpretation cycles may be utilized in conjunction with the present invention. For example, an interpreter may perform dispatch-execute-advance interpretation cycles or there may be more or fewer steps in each cycle. Accordingly, the invention is not limited to the embodiments described herein.

Hybrid Virtual and Native Machine Instructions

In general, the speed in which a program is interpreted can be increased by reducing the average time needed for an interpretation cycle. The invention recognizes the fact that on many modern computers the dispatch step is often the most time consuming step. Accordingly, the advance and dispatch steps of several bytecodes may be combined into a single advance and dispatch step which significantly decreases the execution time needed for such a bytecode sequence.

As an example, assume that the Java source code statement X:=A+B was compiled into bytecodes represented by the following instructions:

Load A

2. Load B

3. Add

4. Store X

The Java virtual machine is a stack based machine. Therefore, after the values of A and B are loaded onto the stack, these values are added and removed from the stack with the result being placed on the stack. The result on the stack is then stored in X and the result is removed from the stack.

When the above virtual machine instructions are interpreted, each of the execute, advance and dispatch steps will take a certain amount of time which may vary from instruction to instruction. Accordingly, the time it takes the interpreter to execute an instruction will be the sum of the time it takes to execute each of the execute, advance and dispatch steps for that instruction. The time it takes to execute a step will be represented by E_n for execute, A_n for advance and D_n for dispatch, where the subscript indicates the number of the instruction to which the time is associated.

The time it takes the interpreter to execute the virtual machine instructions shown above will be the sum of the following times which may occur in this order: E_1, A_1, D_1, E_2, A_2, D_2, E_3, A_3, D_3, E_4, A_4, D_4. With an embodiment of the present invention, a sequence of virtual machine instructions as shown above may be compiled into native machine instructions in the form of a "snippet" so that all but the last advance and dispatch steps may be removed.

As a snippet includes a sequence of native machine instructions, the advance and dispatch steps between the instructions may be eliminated. Therefore, the execution time of the snippet will be approximately the sum of the following times in this order: E_1, E_2, E_3, E_4, A_S, D_S, where the subscript "S" indicates these times represent the snippet advance and dispatch steps which may be different than the traditional advance and dispatch steps. Since the initial advance and dispatch steps are no longer needed to advance the bytecode pointer and fetch the next bytecode, the snippet includes an optimized interpretation cycle for a sequence of bytecodes while preserving interpreter semantics. Conceptually, therefore, a snippet may be considered as an implementation of a higher level bytecode that implements the operations of a sequence of lower level bytecodes.

FIG. **4** shows a high level flowchart of a process of generating a hybrid of virtual and native machine instructions for a function in accordance with one embodiment of the present invention. At step **201**, virtual machine instructions for a function are input into a computer system, such as the ones shown in FIGS. **1** and **2**. In preferred embodiments, the virtual machine instructions for a function are stored as a class file of bytecodes. However, the invention may be readily extended into other interpreted languages by an extension of the principles described herein.

A portion of the virtual machine instructions of the function is selected to be compiled at step **203**. Typically, the system recognizes individual bytecodes or sequences of bytecodes that may be advantageously compiled. For example, the system may generate a snippet for each Java invoke_virtual bytecode that is encountered. Since the invoke_virtual op code may be optimized when it is compiled into native machine instructions within a snippet (see also the section entitled "In-line Caching"). Additionally, statistics may be collected during the interpretation of a program in order to identify portions of the program that would benefit from having a snippet generated.

At step **205**, the selected portion of the function is compiled into one or more native machine instructions. Although snippets usually include more than one native machine instruction, the number of machine instructions is dependent on the virtual machine instructions for which the snippet is replacing.

The virtual machine instruction at the beginning of the selected portion of the function is saved at step **207**. It is not required in all instances that the entire virtual machine instruction be saved. For example, in some embodiments only an initial portion (e.g., first one or more bytes) of a virtual machine instruction are saved. Therefore, when it is stated that a virtual machine instruction is saved, it should be understood that it is meant that at least an initial portion of the virtual machine instruction is saved. Furthermore, in some embodiments more than one virtual machine instruction at the beginning of the selected portion of the function may be saved. It will be readily understood by those of skill in the art that the number of bytes or virtual machine instructions that are saved (or overwritten) may be varied in different embodiments and may depend on the virtual machine instructions themselves.

In order for the snippet to be executed, a new virtual machine instruction (called "go_native" in a preferred embodiment) is executed which specifies the subsequent execution of the snippet. This new virtual machine instruction replaces or overwrites the initial virtual machine instruction of the selected portion of the function. So that the original function or computer program may be restored, the original virtual machine instruction at the beginning of the selected portion is saved, at step **207**, prior to being overwritten. This process will be described in more detail upon reference to FIG. **5** which illustrates how Java virtual machine instructions of a function may be transformed into hybrid virtual and native machine instructions.

At step **209**, the virtual machine instruction at the beginning of the selected portion of the function is overwritten with a new virtual machine instruction that specifies the execution of the one or more native machine instructions of the snippet. In the Java virtual machine, the virtual machine instructions are bytecodes meaning that each virtual machine instruction is composed of one or more bytes. The examples described herein refer to preferred embodiments which increase the execution speed of programs for the Java virtual machine. However, the invention may be advantageously applied to other interpreted languages where the virtual machine instructions may not necessarily be bytecodes.

During execution of an interpreted program, the interpreter decides when to substitute a sequence of bytecodes with a snippet. In a preferred embodiment, if a sequence of bytecodes which may be replaced by a snippet has been found, the interpreter generates a snippet for the sequence and then overwrites the first three bytes of that sequence with a go_native bytecode and a two byte number specifying the snippet. The go_native bytecode is an unused bytecode which is selected for use of the invention.

The snippet will not only hold the native machine instructions, but also the three bytes of the original bytecode that was overwritten as well as a pointer back to their original location so that the snippet may be removed and the original bytecodes restored.

FIG. **5** shows a generation of hybrid virtual and native machine instructions. Java virtual machine instructions **301** are bytecodes where each bytecode may include one or more bytes. The Java virtual machine instructions typically reside in a Java class file as is shown in FIG. **3**. In the example shown, the interpreter decides to introduce a snippet for bytecodes **2–5** of virtual machine instructions **301**. The interpreter generates modified Java virtual machine instructions **303** by overwriting bytecode **2** with a go_native virtual machine instruction.

A snippet zone **305** stores snippets which include native machine instructions. As shown, the go_native bytecode includes a pointer or index to a snippet **307**. Each snippet is a data block that includes two sections of which the first is management information and the second is a sequence of one or more native machine instructions. The management information includes storage for the original bytecode **2** which was overwritten by the go_native bytecode and also the original address of bytecode **2** so that the original bytecode sequence may be restored when the snippet is removed. Typically, the management information section of the snippet is of a fixed length so that the native machine instructions may be easily accessed by a fixed offset. Although snippet **307** is shown as occupying a single "chunk" in snippet zone **305**, snippets may also be allocated that occupy more than one chunk of the snippet zone.

The native machine instruction section of snippet **307** includes native machine instructions for bytecodes **2–5** of virtual machine instructions **301**. Hybrid virtual and native machine instructions **309** include the modified virtual machine instructions and the snippets in the snippet zone. When the interpreter executes the go_native bytecode, the interpreter will look up the snippet in the snippet zone specified by the go_native bytecode and then activate the native machine instructions in the snippet.

The native machine instructions in the snippet perform the same operations as if the bytecodes **2–5** would have been interpreted. Afterwards, the interpreter continues with the execution of bytecode **6** as if no snippet existed. The return of execution in virtual machine instructions **301** is indicated by the dashed arrow in hybrid virtual and native machine instructions **309** shown in FIG. **5**.

The go_native bytecode references (e.g., has a pointer to) the snippet and the snippet includes a reference to the location of the go_native bytecode. The go_native bytecode in a preferred embodiment is 3 bytes long: one for the go_native op code and two bytes for an index into the snippet zone. The two-byte index allows for over 65,000 snippets in the snippet zone. Within the snippet management information section is stored the address of the original bytecode which is currently occupied by the go_native bytecode. This address is utilized to write the original bytecode also stored in the management information section back to its original location. Although a preferred embodiment utilizes a three byte go_native bytecode, there is no requirement that this size be utilized. For example, any number of bytes may be utilized or the size does not have to be limited to byte boundaries.

Snippets should be introduced selectively because it takes time to generate the snippets and because the snippets consume memory space. In a preferred embodiment, snippets are introduced for the following: all or portions of loop bodies, special Java bytecodes (e.g., get_static and put_static), and Java message sends (all the invokexxx bytecodes). In the bytecodes for Java virtual machine, loops are implemented using backward branches. Thus, whenever the interpreter encounters a backward branch, it may introduce a snippet. The snippet generator generates as much native code that will fit into the snippet, starting with the backward branch bytecode. Additionally, some special Java bytecodes and Java message sends may be sped up by using snippets.

FIG. **6** shows a process of introducing a snippet. At step **401**, the system allocates a free snippet in the snippet zone. The free snippet is storage space within the snippet zone which has not been utilized or has been marked as available. One process of allocating a free snippet will be described in more detail in reference to FIG. **7**.

Once a free snippet has been obtained, the one or more virtual machine instructions are compiled into one or more native machine instructions at step **403**. Although the flowcharts show an order to the steps, no specific ordering of the steps should be implied from the figures. For example, it is not necessary that a free snippet be allocated before the virtual machine instructions are compiled into native machine instructions. In fact, in some embodiments it may be beneficial to compile the virtual machine instructions first and then allocate a free snippet to store the native machine instructions, especially if a snippet may span more than one chunk in the snippet zone.

At step **405**, a copy of a selected virtual machine instruction is saved in the management information section of the allocated snippet. The selected virtual machine instruction is the virtual machine instruction that was originally at the beginning of the compiled virtual machine instructions of a function. However, in some embodiments only an initial portion (one or more bytes) of the original virtual machine instruction is saved in the snippet. The address of the original virtual machine instruction in the function is saved in the management information section of the allocated snippet at step **407**.

At step **409**, the original virtual machine instruction is overwritten with a new virtual machine instruction (e.g., go_native) that points to the allocated snippet. As snippets are generated during program execution, the new virtual machine instruction is executed at step **411**.

A snippet may be introduced at arbitrary locations in a bytecode program. However, if the go_native bytecode spans across more than one of the original bytecodes it should be verified that the second and subsequent original bytecodes overwritten by the go_native bytecode are not jump targets or subroutine entry points. More generally, a go_native bytecode should not be used across a basic block entry point. Nevertheless, backward branches as well as many other Java bytecodes are at least three bytes long, thereby providing plenty of storage space for a go_native bytecode. It should be mentioned that a jump to a bytecode after the go_native bytecode which has been compiled into a snippet will not present any problems since the bytecode remains untouched at that location.

Snippets are held and managed in a separate memory space called the snippet zone. The snippet zone may be thought of as a circular list of snippets where a snippet is allocated by either finding an unused snippet in the snippet zone or by recycling a used snippet. Preferably all snippets have the same size to simplify management of the snippet zone. In general, the more snippets that are present in the snippet zone, the longer it will take before a snippet has to be recycled and therefore the faster the program will run.

Now that it has been shown how a snippet may be introduced, FIG. **7** shows a process of allocating a snippet in the snippet zone which was shown as step **401** in FIG. **6**. The process shown in FIG. **7** utilizes a round robin fashion to allocate snippets (i.e., as soon as a new snippet is needed and there are no unused snippets left, the next snippet in the circular list of snippets of the snippet zone is recycled).

At step **501**, the system gets the current snippet. The current snippet in the snippet zone is indicated by a snippet pointer. The system determines if the current snippet is free to be used at step **503**. A flag may be present in the management information section of the snippet to indicate whether the snippet is available. In some embodiments, the field in the management information section of the snippet which stores the address of the original bytecode is set to null if the snippet is available.

If the current snippet is not free, the current snippet is removed at step **505**. Removing the current snippet includes writing the original bytecode stored in the management information section of the snippet to the address of the original bytecode also stored in the management section of the snippet. A process of removing a snippet will be described in more detail in reference to FIG. **9**.

After a snippet has been allocated in the snippet zone, the allocated snippet is set equal to the current snippet, at step **507**, since now the current snippet is free. At step **509**, the system increments the snippet pointer. Although the snippet zone may be thought of as a circular list, the snippet zone may be implemented as an array of chunks. Therefore, if the snippet zone is a linear array, incrementing the snippet pointer may also involve resetting the snippet pointer to the beginning of the snippet zone if the snippet pointer has passed the end of the snippet zone.

Another approach to managing snippets in the snippet zone is to use a time stamp that is stored in the management information section of the snippet indicating the time when the snippet was created or last used. Since it may take substantial resources to find the snippet with the oldest time stamp to be recycled, a combination of time stamps and the round robin fashion may be utilized as follows.

When a free snippet is required, the system may search a predetermined number of snippets after the snippet pointer (e.g., 5 or 10 snippets) in order to locate a snippet with an old time stamp. The snippet with the oldest time stamp near the snippet pointer may then be recycled. Additionally, the time stamp field in the management information section of the snippet may be set to zero or an old time stamp in order to mark the snippet as free.

Now that it has been shown how a snippet may be set up, FIG. **8** shows a process of executing a go_native bytecode. At step **601**, the system gets the snippet index or pointer from the go_native bytecode. The snippet index may be a 2 byte offset into the snippet zone. The system computes the snippet entry point of the native machine instructions within the snippet at step **603**. The snippet entry point is the location of the native machine instructions after the management information section of the snippet. Since the management information section is typically a fixed size, calculating the snippet entry point typically includes adding an offset to the address of the snippet.

The system then jumps to the snippet entry point at step **605** in order to begin execution of the native machine instructions of the snippet. The native machine instructions in the snippet are executed in a step **607**.

Although the implementation of snippets increases the speed of execution of the interpreted code, it is also desirable to provide the capability to reverse the introduction of snippets in order to generate the original bytecodes. For example, after a program in memory has executed, it may be desirable to generate a listing of the original bytecodes without requiring that the original class files be available for access.

FIG. **9** shows a process of removing a snippet in order to produce the original bytecodes. At step **701**, the system replaces the go_native bytecode at the address stored in the management information section of the snippet with the original bytecode (or its initial bytes) also stored in the

management information section. The address stored in the management information section acts as a back pointer to the original bytecode.

Once the original bytecodes are restored, the snippet may be marked as free in the snippet zone at step **703**. The snippet may be marked free in any number of ways depending upon the implementation of the snippet zone. For example, a null pointer may be stored in the address of the original bytecode within the management information section of the snippet. Additionally, if time stamps are being utilized, the time stamp may be set to zero or an old value in order to mark the snippet as free in the snippet zone.

The preceding has described how the invention utilizes dynamically generated native machine instructions for sequences of interpreted code so that a function may be more efficiently executed utilizing a hybrid of virtual and native machine instructions. The execution of an interpreted program can be significantly sped up because frequently used code sequences may be executed in native code rather than an interpreted fashion. The snippets generated are transparent to the interpreter and impose no additional states or complexity. The following will describe implementations of specific virtual machine instruction situations.

In-Line Caching

In the Java virtual machine, the invoke_virtual bytecode is utilized to invoke "normal" functions. The invoke_virtual bytecode includes two bytes which, among other things, specify a function to be invoked. During interpretation of the invoke_virtual bytecode, the interpreter first decodes and executes the invoke_virtual bytecode. The execution of the invoke_virtual bytecode involves fetching the two bytes and determining the starting address of the specified function. However, the determination of the starting address of the specified function may include following multiple levels of pointers to find the class that includes the function. Consequently, the interpretation of an invoke_virtual bytecode maybe be very time consuming.

Snippets may be utilized to expedite the execution of the invoke_virtual bytecode by compiling the invoke_virtual bytecode into the native machine instruction equivalent of "call <function>" (i.e., the starting address of the function is specified without requiring a time consuming search for the starting address of the function). FIG. **10** shows a process of generating a native machine instruction for the invoke_virtual bytecode.

At step **801**, the system finds the function specified in the invoke_virtual bytecode. The process for finding the specified may be the same as is executed by an interpreter (e.g., pointers from class definitions will be followed to find the specified function). Once the specified function is found, the system receives a pointer or address to the specified virtual function at step **803**.

The system then generates native machine instructions for calling the specified virtual function at step **805**. The native machine instructions include the address of the specified function so that execution of the invoke_virtual bytecode will no longer necessitate the time consuming process of finding the starting address of the specified function. By "hard coding" the address of the desired function in native machine instruction, there is a substantial increase in the speed of execution of the invoke_virtual bytecode.

Now that it has been described how the go_native bytecode for implementing the invoke_virtual bytecode has been set up, FIG. **11** shows a process of executing snippet code for the invoke_virtual bytecode. At step **901**, the system saves the current bytecode pointer so that the interpreter can continue at the right location after returning from the function invoked by the invoke_virtual bytecode.

The system pushes the interpreter return address on the stack at step **903**. The interpreter return address is a predefined location where the execution of the interpreter from invoke_virtual bytecodes should resume. The native machine instructions in the snippet for the invoke_virtual function then instruct the system to jump to the function specified in the invoke_virtual bytecodes at step **905**.

Once the virtual function finishes execution, the system returns to the return address that was pushed on the stack at step **907**. At the return address, there are native machine instructions for the interpreter to reload the saved bytecode pointer. At step **909**, recalling that the bytecode pointer was saved at step **901**, the system reloads the saved bytecode pointer so the interpreter may continue where it left off. The interpreter increments the bytecode pointer, at step **909**, in order to indicate the bytecode that should be interpreted next.

As shown above, snippets may be utilized to increase the execution performance of the invoke_virtual bytecode. Other Java bytecodes may be similarly optimized including the invoke_static, invoke_interface, and invoke_special.

Arbitrary Sequences

As described earlier, snippets may be generated for arbitrary sequences of virtual machine instructions. The arbitrary sequences of virtual machine instructions may be selected any number of ways including a statistical analysis that determines execution speed will be increased upon snippetization of the identified sequence of virtual machine instructions.

FIG. **12** shows a process for generating snippet code for an arbitrary sequence of virtual machine instructions. At step **1001**, the system stores the starting bytecode pointer. The starting bytecode pointer indicates the first bytecode that is represented by the snippet that will be generated. At step **1003**, the system sets the current bytecode pointer equal to the starting bytecode pointer. The current bytecode pointer will be utilized to "walk through" the bytecodes as they are compiled and placed in the snippet. The system gets the current bytecode at step **1005**. The current bytecode is specified by the current bytecode pointer.

At step **1007**, the system determines if the snippet has enough room to store the snippet code for the current bytecode and some continuation code. The continuation code is the native machine instructions that implement the equivalent of the advance and fetch steps in the interpretation cycle. If the snippet chunk has enough room, the system determines if a stop snippet flag is set in the bytecode table at step **1009**. The bytecode table is a table maintained by the system to store information about the various bytecodes. This table is shown in FIG. **13** and will be described in more detail later but for the purposes of this flowchart the bytecode table includes a flag which is set in the table for each bytecode to indicate to the system that upon encountering the bytecode, snippet generation should terminate.

At step **1011**, the system emits snippet code (e.g., native machine instructions) specific for the current bytecode. The bytecode specific snippet code may also be stored in the bytecode table as shown in FIG. **13**. The system advances the current bytecode pointer at step **1013**, and then returns to step **1005** to get the next bytecode to analyze.

If snippet generation is to be terminated, the system emits native machine instructions to increment the bytecode pointer at step **1015**. The bytecode pointer should be incremented by the number of byte used by the bytecodes which were placed in the snippet. The system then emits the continuation code at step **1017**. The continuation code is native machine instructions that jump to the address of the

interpreter that interprets the next bytecode. The continuation code may be the same for some bytecodes.

FIG. **13** shows a bytecode table that may be utilized to store information regarding different Java bytecodes. A bytecode table **1051** includes information regarding each of the bytecodes of the virtual machine instructions. In a preferred embodiment, the bytecode table is generated once when the Java virtual machine is initialized. As shown, a bytecode value **1053** (shown in decimal), name of the bytecode **1055** and size of the bytecode **1057** (number of bytes it occupies) are stored in the bytecode table. Additionally, a stop snippet flag **1059** as described in reference to FIG. **12** indicates whether the bytecode should terminate snippet generation when it is encountered.

The bytecode table may include a pointer to snippet code **1061** for each bytecode to which native machine instructions will be generated. Thus, as shown, a template table **1063** may be utilized to store templates for the native machine instructions for each bytecode. The template table allows for fast generation of snippets as the native machine instructions for the bytecodes may be easily determined upon reference to template table **1063**. Additionally, the templates of native machine instructions may also be used to interpret the bytecodes. Another column in bytecode table **1051** may indicate a snippet code size **1065** of the template in the template table.

CONCLUSION

While the above is a complete description of preferred embodiments of the invention, there is alternatives, modifications, and equivalents may be used. It should be evident that the invention is equally applicable by making appropriate modifications to the embodiments described above. For example, the embodiments described have been in reference to increasing the performance of the Java virtual machine interpreting bytecodes, but the principles of the present invention may be readily applied to other systems and languages. Therefore, the above description should not be taken as limiting the scope of the invention which is defined by the metes and bounds of the appended claims along with their full scope of equivalents.

What is claimed is:

1. In a computer system, a method for increasing the execution speed of virtual machine instructions at runtime, the method comprising:

receiving a first virtual machine instruction;

generating, at runtime, a new virtual machine instruction that represents or references one or more native instructions that can be executed instead of said first virtual machine instruction; and

executing said new virtual machine instruction instead of said first virtual machine instruction.

2. The method of claim **1**, further comprising overwriting a selected virtual machine instruction with a new virtual machine instruction, the new virtual machine instruction specifying execution of the at least one native machine instruction.

3. The method of claim **2**, wherein the new virtual instruction includes a pointer to the at least one native machine instruction.

4. The method of claim **2**, further comprising storing the selected virtual machine instruction before it is overwritten.

5. The method of claim **2**, further comprising storing a back pointer to a location of the new virtual machine instruction.

6. The method of claim **2**, wherein the new virtual machine instruction includes a pointer to a data block in which is stored the at least one native machine instruction, a copy of the selected virtual machine instruction, and a back pointer to location of the new virtual machine instruction.

7. The method of claim **6**, further comprising generating the virtual machine instruction that were input by storing the copy of the selected virtual machine instruction stored in the data block at the location specified by the back pointer.

8. In a computer system, a method for increasing the execution speed of virtual machine instructions, the method comprising:

inputting virtual machine instructions for a function;

compiling a portion of the function into at least one native machine instruction so that the function includes both virtual and native machine instruction;

representing said at least one native machine instruction with a new virtual machine instruction that is executed after the compiling of the fuction.

9. A stored data structure of hybrid virtual and native machine instructions, comprising:

a sequence of virtual machine instructions for a function including a new virtual machine instruction;

at least one native machine instruction specified by the new virtual machine instruction for execution with the sequence of virtual machine instructions; and

a computer readable medium that stores the sequence of virtual machine instructions and the at least one native machine instruction;

wherein

the at least one native machine instruction is stored in a data block, and

the data block stores a copy of a selected virtual machine instruction that was overwritten in the sequence of virtual machine instructions by the new virtual machine instruction.

10. The stored data structure of claim **9**, wherein the new virtual machine instruction includes a pointer to the at least one native machine instruction.

11. The stored data structure of claim **9**, wherein the block stores a pointer to a location of the new virtual machine instruction in the sequence of virtual machine instructions.

12. The stored data structure of claim **9**, wherein the data block is stored in an array of blocks.

13. The stored data structure of claim **9**, wherein the at least one native machine instruction is generated from a compilation of a portion of the sequence of virtual machine instructions beginning with the selected virtual machine instruction.

14. The stored data structure of claim **9**, wherein the virtual machine instruction are Java virtual machine instructions.

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 6,910,205 B2
DATED : June 21, 2005
INVENTOR(S) : Bak et al.

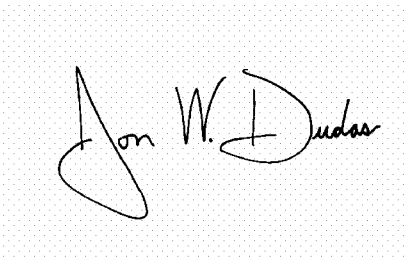
It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

Column 13,
Line 59, change "new virtual" to -- new virtual machine --.

Column 14,
Lines 13 and 58, change "machine instruction" to -- machine instructions --.
Line 25, change "the fuction" to -- the function --.

Signed and Sealed this

Sixth Day of September, 2005



JON W. DUDAS
Director of the United States Patent and Trademark Office

EXHIBIT G

US006061520A

United States Patent [19]

Yellin et al.

[11] **Patent Number: 6,061,520**

[45] **Date of Patent: May 9, 2000**

[54] **METHOD AND SYSTEM FOR PERFORMING STATIC INITIALIZATION**

[75] Inventors: **Frank Yellin**, Redwood City; **Richard D. Tuck**, San Francisco, both of Calif.

[73] Assignee: **Sun Microsystems, Inc.**, Palo Alto, Calif.

[21] Appl. No.: **09/055,947**

[22] Filed: **Apr. 7, 1998**

[51] **Int. Cl.**[7] **G06F 9/45**; G06F 3/00

[52] **U.S. Cl.** **395/705**; 395/704; 395/500.43; 709/100

[58] **Field of Search** 395/705, 707, 395/709

[56] **References Cited**

U.S. PATENT DOCUMENTS

5,361,350	11/1994	Conner et al.	707/103
5,367,685	11/1994	Gosling	395/707
5,421,016	5/1995	Conner et al.	395/707
5,437,025	7/1995	Bale et al.	707/103
5,615,400	3/1997	Cowsar et al.	709/305
5,668,999	9/1997	Gosling	395/704
5,812,828	9/1998	Kaufer et al.	395/500.43
5,815,718	9/1998	Tock	395/705
5,903,899	5/1999	Steele, Jr.	707/206
5,966,702	10/1999	Fresko et al.	707/1
5,999,732	12/1999	Bak et al.	395/705
6,003,038	12/1999	Chen	707/103

OTHER PUBLICATIONS

Tyma, P;, "Tuning Java Performance ". Dr. Dobb's Journal [online], vol. 21, No. 4, pp. 52–58, Apr. 1996.

Cierniak et al., "Briki: an optimizing Java compiler". IEEE/IEEE Electronic Library, Proceedings, IEEE Compcon pp. 179–184, Feb. 1997.

Bell, D.; "Make Java fast: Optimize!". Javaworld[online].

Cramer et al.; "Compiling Java just in time". IEEE Electronic Library[online], vol. 17, Iss. 3, pp. 36–43, May 1997.

Lindholm, Tim et al., The Java Virtual Machine Specification, 1997.

Comar et al.; "Targeting GNAT to the Java virtual machine". ACM Digital Library[online], Proceedings of the conference on TRI–Ada '97, May 1997.

Hsieh et al.; "Compilers for improved Java Performance". IEEE Electronic Library[online]. Computer[online], vol. 30, Iss. 6, pp. 67–75, Jun. 1997.

Armstrong, E.; "Hotspot: A new breed of virtual machine". Javaworld[online].

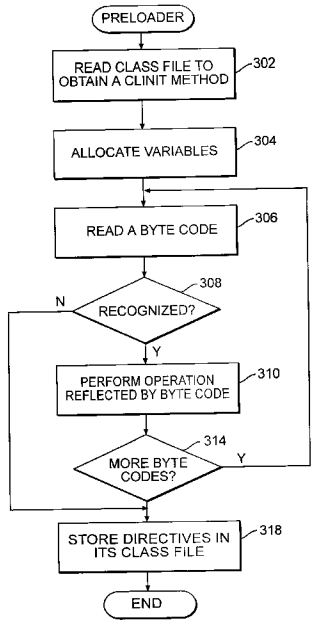
Gosling et al.; The Java Language Specification. Reading, MA, Addison–Wesley. Ch 12, pp. 215–236, Sep. 1996.

Primary Examiner—Tariq R. Hafiz
Assistant Examiner—Kelvin E. Booker
Attorney, Agent, or Firm—Finnegan, Henderson, Farabow, Garrett & Dunner, L.L.P.

[57] **ABSTRACT**

The disclosed system represents an improvement over conventional systems for initializing static arrays by reducing the amount of code executed by the virtual machine to statically initialize an array. To realize this reduction, when consolidating class files, the preloader identifies all <clinit> methods and play executes these methods to determine the static initialization performed by them. The preloader then creates an expression indicating the static initialization performed by the <clinit> method and stores this expression in the .mclass file, replacing the <clinit> method. As such, the code of the <clinit> method, containing many instructions, is replaced by a single expression instructing the virtual machine to perform static initialization, thus saving a significant amount of memory. The virtual machine is modified to recognize this expression and perform the appropriate static initialization of an array.

23 Claims, 3 Drawing Sheets



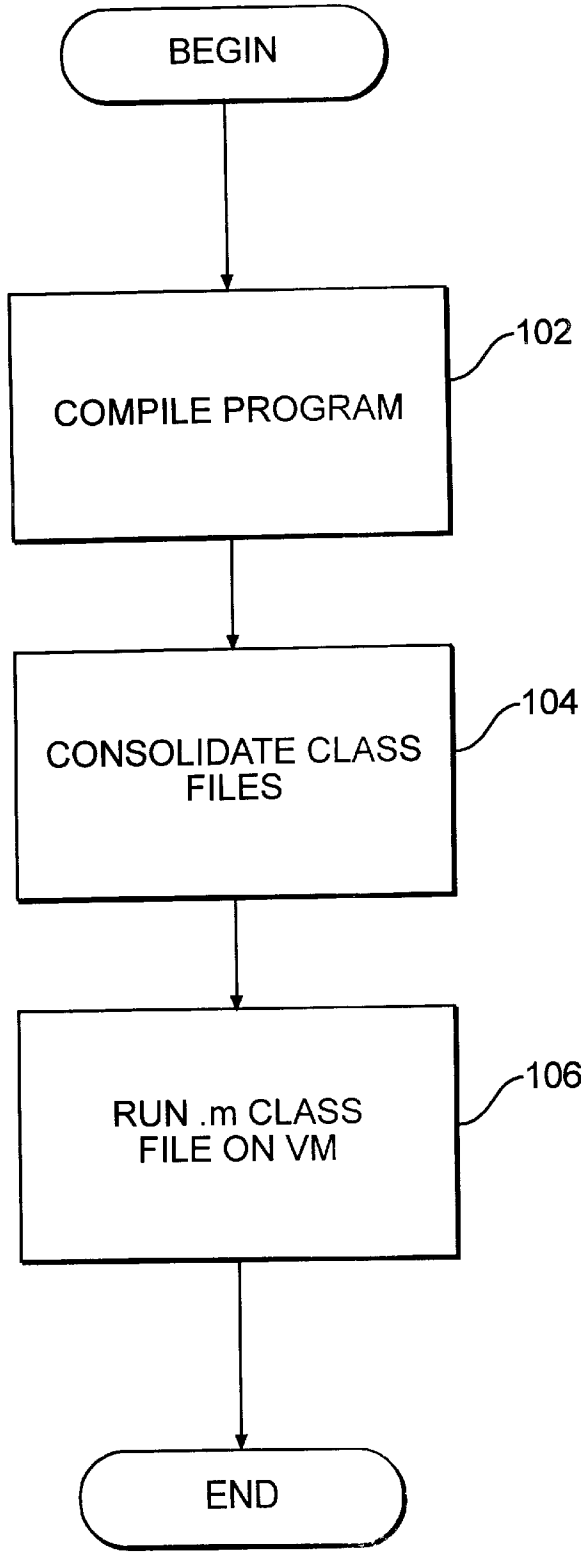


FIG. 1
(PRIOR ART)

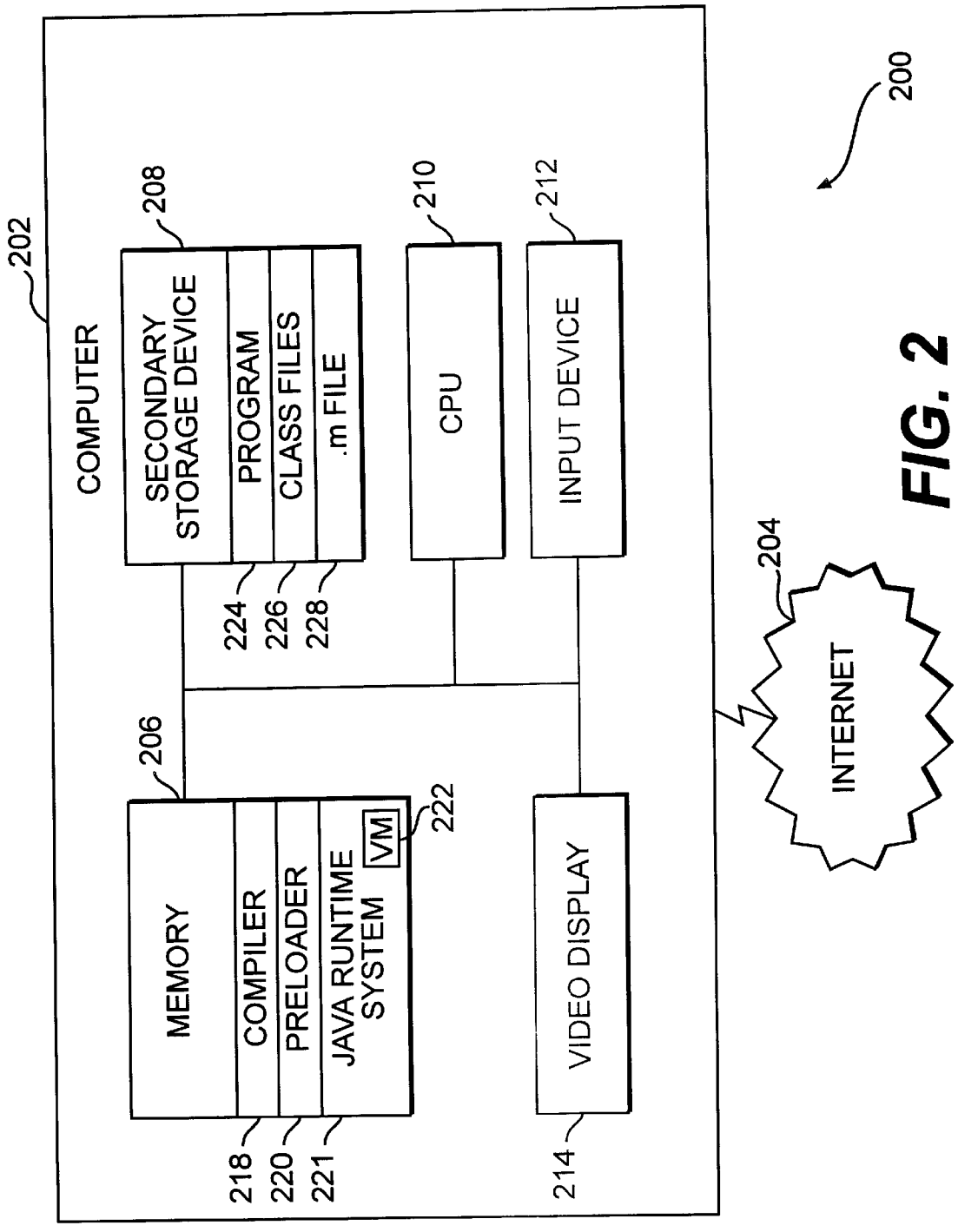

200
202
COMPUTER
208
SECONDARY STORAGE DEVICE
PROGRAM
CLASS FILES
.m FILE
224
226
228
210
CPU
212
INPUT DEVICE
204
INTERNET
206
MEMORY
COMPILER
PRELOADER
JAVA RUNTIME SYSTEM
VM
222
218
220
221
VIDEO DISPLAY
214

FIG. 2

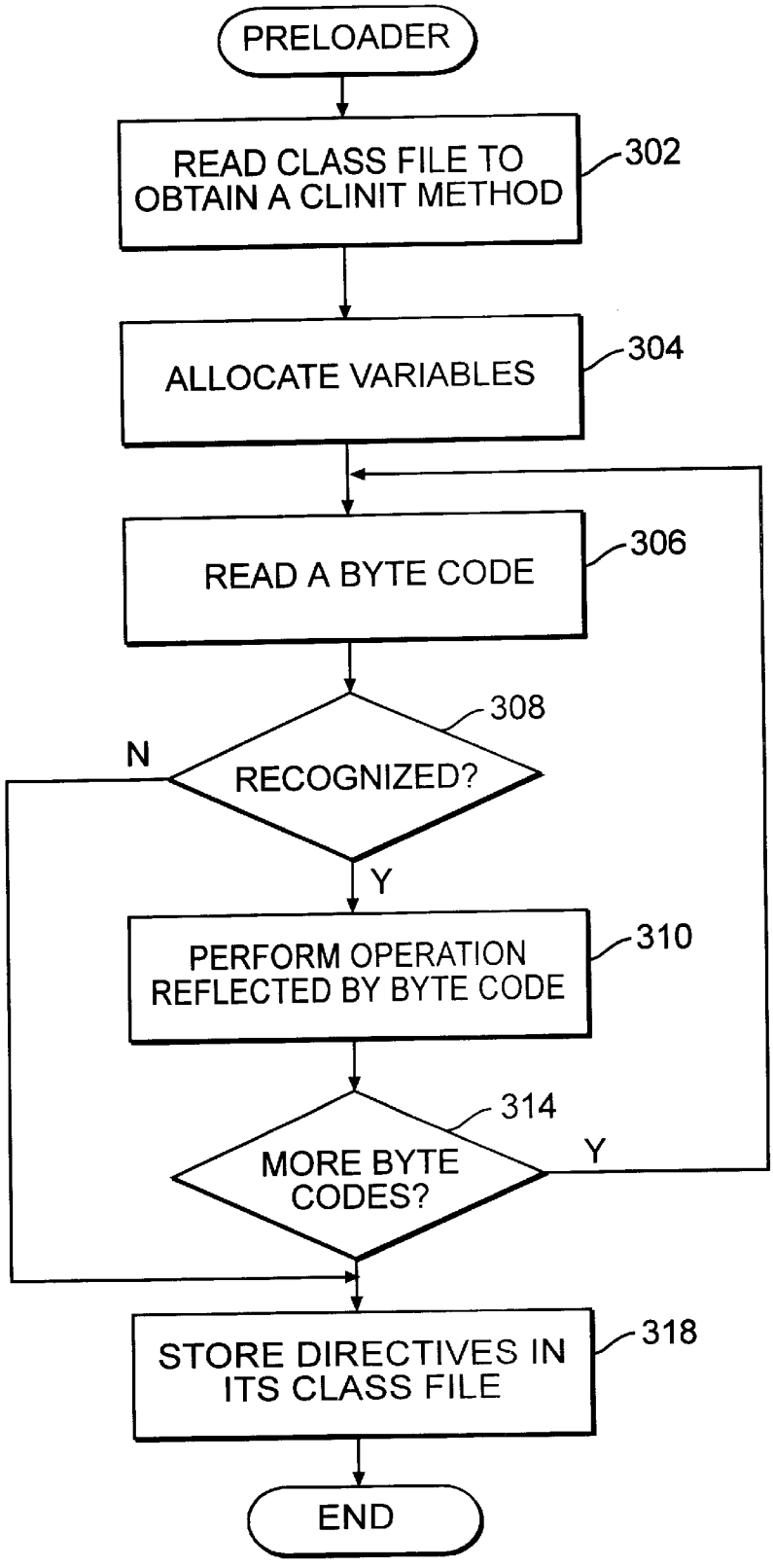

PRELOADER
READ CLASS FILE TO OBTAIN A CLINIT METHOD
302
ALLOCATE VARIABLES
304
READ A BYTE CODE
306
RECOGNIZED?
308
N
Y
PERFORM OPERATION REFLECTED BY BYTE CODE
310
MORE BYTE CODES?
314
Y
STORE DIRECTIVES IN ITS CLASS FILE
318
END

FIG. 3

METHOD AND SYSTEM FOR PERFORMING STATIC INITIALIZATION

FIELD OF THE INVENTION

The present invention relates generally to data processing systems and, more particularly, to a method and system for performing static initialization.

BACKGROUND OF THE INVENTION

Java™ describes both a programming language and a programming environment for generating and running platform-independent code. This platform-independent code runs on a Java™ virtual machine, which is an abstract computing machine that interprets the platform-independent code. The Java™ virtual machine is described in greater detail in Lindholm and Yellin, *The Java Virtual Machine Specification,* Addison-Wesley (1997), which is hereby incorporated by reference. The Java™ virtual machine does not specifically recognize the Java™ programming language or any other programming language; instead, the Java virtual machine only recognizes a particular file format, the class file format. A class file contains the Java virtual machine instructions (or byte codes) that constitute the platform-independent code.

As part of running a Java program, a developer performs a number of steps, as shown in FIG. **1**. First, a developer compiles a computer program (step **102**). Typically, the developer has developed a computer program containing source code in a high-level language, such as the Java programming language, and invokes the Java™ compiler to compile the code. The Java compiler is part of the Java™ software development kit available from Sun Microsystems of Mountain View, Calif,. The Java compiler outputs one or more class files containing byte codes suitable for execution on the Java virtual machine. Each class file contains one type of the Java programming language, either a class or an interface. The class file format is described in greater detail on pp. 83–137 of *The Java Virtual Machine Specification.* Although the class file format is a robust file format, it is unable to instruct the virtual machine to statically initialize an array efficiently, thus posing a problem, discussed in greater detail below.

After compiling the program, the developer consolidates the class files output in step **102** into a single file, known as a .mclass file, by using a preloader (step **104**). The preloader also available from Sun Microsystems, Inc., concatenates the class files and performs preprocessing to facilitate the execution of the class files. After consolidating the class files, the developer loads the .mclass file into a virtual machine (step **106**). In this step, the Java virtual machine stores the .mclass file in memory and interprets the byte codes contained in the .mclass file by reading the byte codes and then processing and executing them. Until interpretation of the byte codes is completed, the .mclass file is stored in memory. The byte codes recognized by the Java virtual machine are more clearly described on pp. 151–338 of *The Java Virtual Machine Specification.*

As stated above, the class file format cannot instruct the virtual machine to statically initialize arrays. To compensate for this problem, the Java™ compiler generates a special method, <clinit>, to perform class initialization, including initialization of static arrays. An example of the initialization of a static array follows:

Code Table #1

```
static int setup[ ]={1, 2, 3, 4};
```

In this example, an array "setup" contains four integers statically initialized to the following values: 1, 2, 3, and 4.

Given this static initialization, the Java™ compiler creates a <clinit> method that performs the static initialization as functionally described below in pseudo-code:

Code Table #2

```
temp=new int [4];
temp=[0]=1;
temp=[1]=2;
temp=[2]=3;
temp=[3]=4;
this.setup=temp;
```

As the above code table shows, merely describing the <clinit> method functionally requires a number of statements. More importantly, however, the actual processing of the <clinit> method, performed by byte codes, requires many more statements. These byte codes manipulate a stack resulting in the requested static initialization. A stack is a portion of memory used by the methods in the Java programming environment. The steps performed by the <clinit> method for the example static initialization described above are expressed below in byte codes.

Code Table #3

```
Method void <clinit>()
0 iconst_4 //push an integer value of 4 on the stack
1 newarray int //create a new array of integers and put it
    on the stack.
3 dup //duplicate top of stack
4 iconst_0 //push an integer value of 0 on the stack
5 iconst_1 //push an integer value of 1 on the stack
6 iastore //store a 1 at index 0 of array
7 dup //duplicate the top of the stack
8 iconst_1 //push an integer value of 1 on the stack
9 iconst_2 //push an integer value of 2 on the stack
10 iastore //store a 2 at index 1 of array
11 dup //duplicate top of stack
12 iconst_2 //push an integer value of 2 on the stack
13 iconst_3 //push an integer value of 3 on the stack
14 iastore //store a 3 at index 2 of array
15 dup //duplicate top of stack
16 iconst_3 //push an integer of value 3 on stack
17 iconst_4 //push an integer of value 4 on stack
18 iastore //store a 4 at index 3 of array
19 putstatic #3<Field foobar.setup [I> //modify set up
    array according to new array on stack
22 return
```

Although using the <clinit> method provides the Java™ compiler with a way to instruct the virtual machine to initialize a static array, the amount of code required to initialize the array is many times the size of the array, thus requiring a significant amount of memory. It is therefore desirable to improve static initialization.

SUMMARY OF THE INVENTION

The disclosed system represents an improvement over conventional systems for initializing static arrays by reducing the amount of code executed by the virtual machine to statically initialize an array. To realize this reduction, when consolidating class files, the preloader identifies all <clinit> methods and simulates executing ("play executes") these methods to determine the static initialization performed by

them. The preloader then creates an expression indicating the static initialization performed by the <clinit> method and stores this expression in the .mclass file, replacing the <clinit> method. As such, the code of the <clinit> method, containing many instructions, is replaced by a single expression instructing the virtual machine to perform static initialization, thus saving a significant amount of memory. The virtual machine is modified to recognize this expression and perform the appropriate static initialization of an array.

Methods consistent with the present invention receive code to be run on a processing component to perform an operation. The code is then play executed on the memory without running the code on the processing component to identify the operation if the code were run by the processing component. Thereafter, a directive is created for the processing component to perform the operation.

A data processing system consistent with the present invention contains a secondary storage device, a memory, and a processor. The secondary storage device contains a program with source code that statically initializes the data structure and class files, where one of the class files contains a <clinit> method that statically initializes the data structure. The memory contains a compiler for compiling the program and for generating the class files and a preloader for consolidating the class files, for simulating execution of the <clinit> method to determine the static initialization the <clinit> method performs, and for creating an instruction to perform the static initialization. The processor runs the compiler and the preloader.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. **1** depicts a flowchart of the steps performed when developing a program in the Java™ programming environment.

FIG. **2** depicts a data processing system consistent with the present invention.

FIG. **3** depicts a flowchart of the steps performed by the preloader depicted in FIG. **2**.

DETAILED DESCRIPTION OF THE INVENTION

Systems and methods consistent with the present invention provide an improved system for initializing static arrays in the Java™ programming environment by replacing the <clinit> method with one or more directives which, when read by the virtual machine, causes the virtual machine to perform the same static initialization performed by the <clinit> method, except using a significantly less amount of memory and significantly less time. As a result, such systems and methods can significantly reduce memory utilization when statically initializing an array.

Overview

Systems and methods consistent with the present invention eliminate the need for the <clinit> method by performing certain preprocessing in the preloader. Specifically, the preloader receives class files for consolidation and scans them looking for a <clinit> method. When the preloader finds the <clinit method, it simulates execting ("play executes") the <clinit> method against memory to determine the effects that the <clinit> method would have on the memory if interpreted by the Java virtual machine. That is, the preloader simulates execution of the <clinit> method to identify the static initialization that would result had the <clinit> method been executed by the Java™ virtual machine. After identifying this static initialization, the preloader generates one or more directives (or instructions) to cause the same static initialization as the <clinit> method and outputs these directives to the Java virtual machine, thus replacing the <clinit> method. These directives are then read at runtime by the Java virtual machine causing the Java virtual machine to perform the same static initialization performed by the <clinit> method. The directives require significantly less memory space than the <clinit> method. For example, the byte codes described above in code table #3 could be reduced to the following directives contained within the .mclass file indicating that an array of four integers has the initial values 1, 2, 3, and 4:

CONSTANT_Array T_INT 4 1 2 3 4

The virtual machine of an exemplary embodiment recognizes this expression and statically initializes the array to the appropriate values. As a result, the exemplary embodiment reduces memory consumption over conventional systems when initializing a static array.

Implementation Details

FIG. **2** depicts a data processing system **200** consistent with the present invention. The data processing system **200** comprises a computer system **202** connected to the Internet **204**. Computer system **202** contains a memory **206**, a secondary storage device **208**, a central processing unit (CPU) **210**, an input device **212**, and a video display **214**. The memory **206** further includes the Java™ compiler **218**, the Java™ preloader **220**, and the Java™ runtime system **221**. The Java™ runtime system **221** includes the Java™ virtual machine **222**. The secondary storage device **208** contains a program **224** with source code, various class files **226**, and a .mclass file **228**. The Java™ compiler **218** compiles the program **224** into one or more class files **226**. The preloader **220** then receives the class files **226** and generates a .mclass file **228** representing the consolidation of all of the class files. After consolidation, the .mclass file **228** can be run on the virtual machine **222**.

Processing consistent with the present invention is performed by the preloader **220** searching for a <clinit> method, and when it is found, the preloader (1) simulates execution of the <clinit> method to determine the effects it would have on memory if it was interpreted by the virtual machine **222**, (2) creates static initialization directives to replicate these effects, and (3) outputs these directives in the .mclass file to replace the <clinit> method, thus saving significant amounts of memory.

In addition, processing consistent with the present invention is performed by the virtual machine **222** because it is modified to recognize the static initialization directives of the preloader. Although an exemplary embodiment of the present invention is described as being stored in memory **206**, one skilled in the art will appreciate that it may also be stored on other computer-readable media, such as secondary storage devices like hard disks, floppy disks, or CD-Rom; a carrier wave received from the Internet **204**; or other forms of RAM or ROM. Additionally, one skilled in the art will appreciate that computer **202** may contain additional or different components.

The Preloader

FIG. **3** depicts a flowchart of the steps performed by the preloader **220** consistent with the present invention to perform initialization of a static array. The first step performed by the preloader is to read a class file to obtain the <clinit> method (step **302**). After obtaining a <clinit> method, the preloader allocates various variables for use during play execution (step **304**). When play executing, discussed below, the preloader simulates execution of the byte codes contained in the <clinit> method by the virtual machine. These byte codes manipulate various data structures associated

with the <clinit> method, such as the constant pool, the stack, or local variables (or registers).

The constant pool is a table of variable-length structures representing various string constants, class names, field names, and other constants referred to within the class file. The stack is a portion of memory for use in storing operands during the execution of the method. Thus, the size of the stack is the largest amount of space occupied by the operands at any point during execution of this method. The local variables are the variables that are used by this method.

When allocating variables, the preloader obtains a pointer to the constant pool of the <clinit> method, allocates a stack to the appropriate size, and allocates an array such that one entry of the array corresponds to each of the local variables. As described below, the play execution operates on these variables.

After allocating the variables, the preloader reads a byte code from the <clinit> method (step **306**). Next, the preloader determines if it recognizes this byte code (step **308**). In this step, the preloader recognizes a subset of all byte codes where this subset contains only those byte codes that are generally used to perform static initialization of an array. Following is a list of the byte codes recognized by the preloader of an exemplary embodiment:

Code Table #4

aconst_null	iastore
iconst_m1	lastore
iconst_0	fastore
iconst_1	dastore
iconst_2	aastore
iconst_3	bastore
iconst_4	lastore
iconst_5	sastore
lconst_0	dup
lconst_1	newarray
fconst_0	anewarray
fconst_1	return
fconst_2	ldc
dconst_0	ldc_w
dconst_1	ldc2_w
bipush	putstatic
sipush	

Any byte codes other than those listed above are not recognized. The appearance of other byte codes beyond those described above indicates that the <clinit> method performs functionality in addition to statically initializing an array. In this case, the <clinit> method cannot be optimized. If a byte code is not recognized, the preloader considers it unsuitable for optimization (or play execution) and processing continues to step **316**.

If the preloader recognizes the byte code, however, the preloader performs the operation reflected by the byte code (step **310**). In this step, the preloader play executes the byte code on the variables allocated in step **304**, and as a result, a value may be popped from the stack, a local variable may be updated, or a value from the constant pool may be retrieved. Additionally, the preloader may encounter a "put static" byte code indicating that a particular static variable (e.g., array) is to be initialized in a particular manner. If the preloader receives such a byte code, it stores an indication of the requested initialization into a hash table for later use. An example of such an entry in the hash table follows:

Setup:=Array (1,2,3,4)

After performing the operation reflected by the byte code, the preloader determines if there are more byte codes in the <clinit> method (step **314**). If so, processing returns to step **306**. However, if there are no more byte codes, the preloader stores directions in the .mclass file to statically initialize the arrays (step **318**). In this step, the preloader stores constant pool entries into the .mclass file like the following:

Tag	Type	Size	Values
CONSTANT_Array	T_INT	4	1 2 3 4

This entry in the constant pool indicates that a particular array has four integers that have the initial values of 1, 2, 3, and 4. At run time, when the virtual machine initializes the class .mclass file, it will encounter a reference to this constant pool entry and create the appropriate array. As a result, the many instructions contained in the <clinit> method are reduced to this one expression, saving significant amounts of memory and time.

Example Implementation of the Preloader

The following pseudo-code describes sample processing of the preloader of an exemplary embodiment. The preloader receives as a parameter a method information data structure that defines the <clinit> method, described in the *Java™ Virtual Machine Specification* at pp. 104–106, and play executes the byte codes of this <clinit> method. It should be noted that the processing described is only exemplary; as such, only a few byte codes are described as being processed by the preloader. However, one skilled in the art will appreciate that all of the byte codes in code table #4 may be processed by the exemplary embodiment.

Code Table #5

```
void emulateByteCodes(Method_info mb)
  int numberRegisters = mb.max_locals();    //number of local variables
  int stackSize = mb.max_stack();           //stack size
  byte byteCode [] = mb.code();             //get the byte code
  ConstantPool constantPool = mb.constantPool();   // get constant pool
  Object stack[] = new Object[stackSize];          //create stack for
                                                     play execution
  Object registers[] = new Object[numberRegisters];  //create local
                                                       variables
                                                       for play
                                                         //execution
  /* Start with an empty stack. */
  int stackTop = -1;                        //just below valid element
  /* Map of static objects */
  Hashtable changes = new Hashtable();
try {
    boolean success;
  execution_loop:
    for (int codeOffset = 0, nextCodeOffset;
         ;codeOffset = nextCodeOffset) {
      int opcode = byteCode[codeOffset] & 0xFF;      //0..255
      nextCodeOffset = codeOffset + 1;   // the most usual value
      switch(opcode) {
        case opc_iconst_m1:   // push -1 on the stack
          stack[++stackTop] = new Integer(-1);
          break;
        case opc_bipush:
          nextCodeOffset = codeOffset + 2;
          stack[++stackTop] = new Integer(bytecode[codeOffset + 1]);
          break;
        case opc_1load_3:          //load the contents of register 3
          stack[++stackTop] = (Long)register [3];
          stack[++stackTop] = null;       //longs use two words on stack
          break;
        case opc_fsub: {          // subtract top of stack from item below
          float b = stack[stackTop--].floatValue();
          float a = stack[stackTop].floatValue();
```

-continued

Code Table #5

```
            stack[stackTop] = new Float(a - b);
            break;
          }
          case opc_1dc:
            nextCodeOffset = codeOffset + 2;
            stack[++stackTop] =
                constantPool.getItem(byteCode (codeOffset + 1));
            break;
          case sastore: {// store the contents into a "short" array
            short value = (short) (stack[StackTop--].intValue());
            int index = stack[StackTop--].intvalue();
            short[] array = (short[])stack[StackTop--];
            array[index] = value;
            break;
          }
          case opc_putstatic: {
            nextCodeOffset = codeOffset + 3;
            int index = ((byteCode[codeOffset + 1]) & 0xFF) << 8) +
              (byteCode[codeOffset + 2] & 0xFF);
            Field f = constantPool.getItem(byteCode[codeOffset + 1]);
            if (f.getClass() ! = mb.getClass()) {
              // we can only modify static's in our own class
              throw new RuntimeException();
            }
            Type t = f.getType();
            if (t.isLong() || t.isDouble())
              ++stackTop;
            Object value = stack[++stackTop]
            changes.put(f, value); // put entry into hashtable
            break;
          case opc_return:
            success = true;
            break execution_loop;
          default: // some byte code we do not understand
            success = false;
            break execution_loop;
          }
        }
      } catch (RuntimeException) {
        // any runtime exception indicates failure.
        success = false;
      }
      if (success) {
        <modify .class file as indicated by "changes" hashtable>
        <Remove this <clinit> method from the class>
      } else {
        <ran into something we cannot understand>
        <do not replace this method>
    }
}
```

The Virtual Machine of the Exemplary Embodiment

As stated above, the Java virtual machine **222** is an otherwise standard Java virtual machine as defined in the *Java Virtual Machine Specification,* except that it is modified as will be described below. Conventional virtual machines recognize various constant pool entries, such as CONSTANT_Integer, CONSTANT_String, and CONSTANT_Long. Constant pool entries of these types store various variable information, including the initial value. The virtual machine of an exemplary embodiment, however, additionally recognizes the CONSTANT_Array entry in the constant pool.

The format of the CONSTANT_Array constant pool entry in the class file format follows:

Code Table #6

```
CONSTANT_Array_info {
  u1 tag;                  /* The literal value CONSTANT_Array */
```

-continued

Code Table #6

```
  u1 type;                 /* see below */
  u4 length;               /* number of elements of the array */
  ux objects[length];      /* Actual values */
      /* The following field is included only if type ==
      T_CLASS */
  u2 type2;                /* index of CONSTANT_Class in constant pool */
}
```

The u1 type field is one of the values listed in the following table:

Array Type	Value
T_CLASS	2
T_BOOLEAN	4
T_CHAR	5
T_FLOAT	6
T_DOUBLE	7
T_BYTE	8
T_SHORT	9
T_INT	10
T_LONG	11

The field ux objects[length] is an array of values, providing the elements of the array. The number of elements in the array is given by the length field of the constant pool entry. The actual size of each of these values is shown below:

Type	ux	Meaning
T_BOOLEAN, T_BYTE	u1	1 byte
T_CHAR, T_SHORT, T_CLASS	u2	2 bytes
T_INT, T_FLOAT	u4	4 bytes
T_LONG, T_DOUBLE	u8	8 bytes

For all of the above types except for T_CLASS, the bytes shown are the actual value that are stored in that element of the array. For T_CLASS, however, each u**2** is itself an index to an entry into the constant pool. The constant pool entry referred to must itself be either a CONSTANT_Array, CONSTANT_Object, or the special constant pool entry 0, indicating a NULL value.

For example, to indicate the following array:

```
int[ ]={10, 20, 30, 40 };
```

the constant pool entry would be as follows:

Tag	Type	Size	Initial Values
CONSTANT_Array	T_INT	4	10 20 30 40

As another example, to indicate the following array:

```
new Foo[3 ]/* all initialized to NULL */
```

the constant pool entry would be as follows:

Tag	Type	Size	Initial Values	Class
CONSTANT_Array	T_CLASS	3	0 0 0	XX

where "xx" is an index into the constant pool indicating the class Foo in the constant pool.

Two-dimensional arrays like the following:

```
new byte[ ][ ]={{1,2,3,4 }, {5,6,7,8 }};
```

are encoded by having two constant pool entries encode the sub-arrays and by having two additional entries indicate the association between the subarrays. This encoding corresponds to the Java™ notion of an array as a type of object and a multi-dimensional array as an array of arrays. The constant pool entries of the above two-dimensional array follows:

Entry1: CONSTANT_Array T_BYTE **4 1 2 3 4**

Entry2: CONSTANT_Array T_BYTE **4 5 6 7 8**

Entry3: CONSTANT_Class with name "[[B"

and then

	Tag	Type	Size	Initial Values	Class
Entry4:	CONSTANT_Array	T_Class	2	Entry1 Entry2	Entry3

where each of Entry1, Entry2, and Entry3 are the two-byte encodings of the index of the corresponding constant-pool entry.

While the systems and methods of the present invention have been described with reference to a preferred embodiment, those skilled in the art will know of various changes in form and detail which may be made without departing from the spirit and scope of the present invention as defined in the appended claims.

What is claimed is:

1. A method in a data processing system for statically initializing an array, comprising the steps of:

compiling source code containing the array with static values to generate a class file with a clinit method containing byte codes to statically initialize the array to the static values;

receiving the class file into a preloader;

simulating execution of the byte codes of the clinit method against a memory without executing the byte codes to identify the static initialization of the array by the preloader;

storing into an output file an instruction requesting the static initialization of the array; and

interpreting the instruction by a virtual machine to perform the static initialization of the array.

2. The method of claim **1** wherein the storing step includes step of:

storing a constant pool entry into the constant pool.

3. The method of claim **1** wherein the play executing step includes the steps of:

allocating a stack;

reading a byte code from the clinit method that manipulates the stack; and

performing the stack manipulation on the allocated stack.

4. The method of claim **1** wherein the play executing step includes the steps of:

allocating variables;

reading a byte code from the clinit method that manipulates local variables of the clinit method; and

performing the manipulation of the local variables on the allocated variables.

5. The method of claim **1** wherein the play executing step includes the steps of:

obtaining a reference to a constant pool of the clinit method;

reading a byte code from the clinit method that manipulates the constant pool; and

performing the constant pool manipulation.

6. A method in a data processing system, comprising the steps of:

receiving code to be run on a processing component to perform an operation;

play executing the code without running the code on the processing component to identify the operation if the code were run by the processing component; and

creating an instruction for the processing component to perform the operation.

7. The method of claim **6** wherein the operation initializes a data structure, and wherein the play executing step includes the step of:

play executing the code to identify the initialization of the data structure.

8. The method of claim **6** wherein the operation statically initializes an array and wherein the play executing step includes the step of:

play executing the code to identify the static initialization of the array.

9. The method of claim **6** further including the step of:

running the created instruction on the processing component to perform the operation.

10. The method of claim **6** further including the step of:

interpreting the created instruction by a virtual machine to perform the operation.

11. The method of claim **6** wherein the operation has an effect on memory, and wherein the play executing step includes the step of:

play executing the code to identify the effect on the memory.

12. A data processing system comprising:

a storage device containing:

a program with source code that statically initializes a data structure; and

class files, wherein one of the class files contains a clinit method that statically initializes the data structure;

a memory containing:

a compiler for compiling the program and generating the class files; and

a preloader for consolidating the class files, for play executing the clinit method to determine the static initialization the clinit method performs, and for creating an instruction to perform the static initialization; and

a processor for running the compiler and the preloader.

13. The data processing system of claim 12 wherein the preloader includes a mechanism for generating an output file containing the created instruction.

14. The data processing system of claim 13 wherein the memory further includes a virtual machine that interprets the created instruction to perform the static initialization.

15. The data processing system of claim 12, wherein the data structure is an array.

16. The data processing system of claim 12 wherein the clinit method has byte codes that statically initialize the data structure.

17. The data processing system of claim 12 wherein the created instruction includes an entry into a constant pool.

18. A computer-readable medium containing instructions for controlling a data processing system to perform a method, comprising the steps of:

receiving code to be run on a processing component to perform an operation;

simulating execution of the code without running the code on the processing component to identify the operation if the code were run by the processing component; and

creating an instruction for the processing component to perform the operation.

19. The computer-readable medium of claim 18 wherein the operation initializes a data structure, and wherein the simulating step includes the step of:

simulating execution of the code to identify the initialization of the data structure.

20. The computer-readable medium of claim 18 wherein the operation statically initializes an array and wherein the simulating step includes the step of:

simulating execution of the code to identify the static initialization of the array.

21. The computer-readable medium of claim 18 further including the step of:

running the created instruction on the processing component to perform the operation.

22. The computer-readable medium of claim 18 further including the step of:

interpreting the created instruction by a virtual machine to perform the operation.

23. The computer-readable medium of claim 18 wherein the operation has an effect on memory, and wherein the simulating step includes the step of:

simulating execution of the code to identify the effect on the memory.

* * * * *

EXHIBIT H

Certificate of Registration

This Certificate issued under the seal of the Copyright Office in accordance with title 17, United States Code, attests that registration has been made for the work identified below. The information on this certificate has been made a part of the Copyright Office records.

Marybeth Peters

Register of Copyrights, United States of America

Form TX
For a Nondramatic Literary Work
UNITED STATES COPYRIGHT OFFICE

TX 6-196-514

EFFECTIVE DATE OF REGISTRATION
04 20 2005
Month Day Year

DO NOT WRITE ABOVE THIS LINE. IF YOU NEED MORE SPACE, USE A SEPARATE CONTINUATION SHEET

1 TITLE OF THIS WORK ▼

Java 2 Standard Edition 1.4

PREVIOUS OR ALTERNATIVE TITLES ▼
J2SE 1.4, Java 2 Platform, Standard Edition, v 1.4, Java 2 Standard Edition Software Development Kit 1.4, SDK 1.4

PUBLICATION AS A CONTRIBUTION If this work was published as a contribution to a periodical, serial, or collection, give information about the collective work in which the contribution appeared. Title of Collective Work ▼

If published in a periodical or serial give: Volume ▼ Number ▼ Issue Date ▼ On Pages ▼

2 a NAME OF AUTHOR ▼
Sun Microsystems, Inc

DATES OF BIRTH AND DEATH
Year Born ▼ Year Died ▼

Was this contribution to the work a "work made for hire"?
☑ Yes
☐ No

AUTHOR'S NATIONALITY OR DOMICILE
Name of Country
OR { Citizen of ▶ ____
Domiciled in ▶ United States

WAS THIS AUTHOR'S CONTRIBUTION TO THE WORK
Anonymous? ☐ Yes ☑ No
Pseudonymous? ☐ Yes ☑ No
If the answer to either of these questions is "Yes," see detailed instructions.

NATURE OF AUTHORSHIP Briefly describe nature of material created by this author in which copyright is claimed ▼
New and revised computer code and accompanying documentation and manuals

NOTE
Under the law, the "author" of a "work made for hire" is generally the employer, not the employee (see instructions). For any part of this work that was "made for hire" check "Yes" in the space provided, give the employer (or other person for whom the work was prepared) as "Author" of that part, and leave the space for dates of birth and death blank.

b NAME OF AUTHOR ▼
(SEE FORM TX/CON FOR ADDITIONAL AUTHORS)

DATES OF BIRTH AND DEATH
Year Born ▼ Year Died ▼

Was this contribution to the work a "work made for hire"?
☐ Yes
☐ No

AUTHOR'S NATIONALITY OR DOMICILE
Name of Country
OR { Citizen of ▶ ____
Domiciled in ▶ ____

WAS THIS AUTHOR'S CONTRIBUTION TO THE WORK
Anonymous? ☐ Yes ☐ No
Pseudonymous? ☐ Yes ☐ No
If the answer to either of these questions is "Yes," see detailed instructions.

NATURE OF AUTHORSHIP Briefly describe nature of material created by this author in which copyright is claimed ▼

c NAME OF AUTHOR ▼

DATES OF BIRTH AND DEATH
Year Born ▼ Year Died ▼

Was this contribution to the work a "work made for hire"?
☐ Yes
☐ No

AUTHOR'S NATIONALITY OR DOMICILE
Name of Country
OR { Citizen of ▶ ____
Domiciled in ▶ ____

WAS THIS AUTHOR'S CONTRIBUTION TO THE WORK
Anonymous? ☐ Yes ☐ No
Pseudonymous? ☐ Yes ☐ No
If the answer to either of these questions is "Yes," see detailed instructions.

NATURE OF AUTHORSHIP Briefly describe nature of material created by this author in which copyright is claimed ▼

3 a YEAR IN WHICH CREATION OF THIS WORK WAS COMPLETED This information must be given in all cases.
*-2004-2002 ◀ Year

b DATE AND NATION OF FIRST PUBLICATION OF THIS PARTICULAR WORK
Complete this information ONLY if this work has been published.
Month ▶ February Day ▶ 13 Year ▶ 2002
United States ◀ Nation

4 COPYRIGHT CLAIMANT(S) Name and address must be given even if the claimant is the same as the author given in space 2 ▼
Sun Microsystems, Inc
4150 Network Circle
Santa Clara, CA 95054

See instructions before completing this space

TRANSFER If the claimant(s) named here in space 4 is (are) different from the author(s) named in space 2, give a brief statement of how the claimant(s) obtained ownership of the copyright. ▼
By written agreement

DO NOT WRITE HERE OFFICE USE ONLY

APPLICATION RECEIVED
APR 20 2005

ONE DEPOSIT RECEIVED
APR 20 2005

TWO DEPOSITS RECEIVED

FUNDS RECEIVED

MORE ON BACK ▶ Complete all applicable spaces (numbers 5-9) on the reverse side of this page. See detailed instructions. Sign the form at line 8.

DO NOT WRITE HERE

*Amended by C O Authority of Susanne Morales from phone call on 07/26/2005

EXAMINED BY

CHECKED BY

FORM TX

☐ CORRESPONDENCE Yes

FOR COPYRIGHT OFFICE USE ONLY

DO NOT WRITE ABOVE THIS LINE IF YOU NEED MORE SPACE, USE A SEPARATE CONTINUATION SHEET

PREVIOUS REGISTRATION Has registration for this work, or for an earlier version of this work, already been made in the Copyright Office? **5**

☑ Yes ☐ No If your answer is 'Yes' why is another registration being sought? (Check appropriate box) ▼

a ☐ This is the first published edition of a work previously registered in unpublished form

b ☐ This is the first application submitted by this author as copyright claimant

c ☑ This is a changed version of the work, as shown by space 6 on this application.

(See Form TX/CON for Previous Registrations)

If your answer is "Yes" give Previous Registration Number ▶ Year of Registration ▶

DERIVATIVE WORK OR COMPILATION **6**

Preexisting Material Identify any preexisting work or works that this work is based on or incorporates ▼ a

Prior works by claimant and licensed-in components

See instructions before completing this space

Material Added to This Work Give a brief, general statement of the material that has been added to this work and in which copyright is claimed ▼ b

New and revised computer code and accompanying documentation and manuals

DEPOSIT ACCOUNT If the registration fee is to be charged to a Deposit Account established in the Copyright Office give name and number of Account **7**

Name ▼ Account Number ▼ a

CORRESPONDENCE Give name and address to which correspondence about this application should be sent. Name/Address/Apt/City/State/ZIP ▼ b

Ines Gonzalez, Esq
Fenwick & West LLP
801 California Street
Mountain View CA 94041

Area code and daytime telephone number ▶ (650) 335 7182

Fax number ▶ (650) 938 5200

Email ▶ igonzalez@fenwick com

CERTIFICATION* I, the undersigned, hereby certify that I am the **8**

Check only one ▶
- ☐ author
- ☐ other copyright claimant
- ☐ owner of exclusive right(s)
- ☑ authorized agent of Sun Microsystems Inc

Name of author or other copyright claimant, or owner of exclusive right(s) ▲

of the work identified in this application and that the statements made by me in this application are correct to the best of my knowledge

Typed or printed name and date ▼ If this application gives a date of publication in space 3 do not sign and submit it before that date

Marilyn E Glaubensklee, Assistant General Counsel Date ▶ 4/15/05

Handwritten signature (X) ▼

X Marilyn E Glaubensklee

Certificate will be mailed in window envelope to this address

Name ▼
Susanne S Morales Paralegal / Fenwick & West LLP

Number/Street/Apt ▼
801 California Street

City/State/ZIP ▼
Mountain View, CA 94041

9

Complete all necessary spaces
Sign your application in space 8

1 Application form
2. Nonrefundable filing fee in check or money order payable to *Register of Copyrights*
3. Deposit material

Library of Congress
Copyright Office TX
101 Independence Avenue S E
Washington D C 20559-6222

Fees are subject to change. For current fees, check the Copyright Office website at www.copyright.gov, write the Copyright Office, or call (202) 707-3000.

17 U.S.C. § 506(e) Any person who knowingly makes a false representation of a material fact in the application for copyright registration provided for by section 409 or in any written statement filed in connection with the application shall be fined not more than $2,500

CONTINUATION SHEET FOR APPLICATION FORMS

Form TX /CON
UNITED STATES COPYRIGHT OFFICE

- This Continuation Sheet is used in conjunction with Forms CA, PA, SE SR TX and VA only Indicate which basic form you are continuing in the space in the upper right-hand corner
- If at all possible try to fit the information called for into the spaces provided on the basic form
- If you do not have enough space for all the information you need to give on the basic form use this Continuation Sheet and submit it with the basic form
- If you submit this Continuation Sheet clip (do not tape or staple) it to the basic form and fold the two together before submitting them
- Space A of this sheet is intended to identify the basic application
  Space B is a continuation of Space 2 on the basic application
  Space B is not applicable to Short Forms
  Space C (on the reverse side of this sheet) is for the continuation of Spaces 1 4, or 6 on the basic application or for the continuation of Space 1 on any of the three Short Forms PA, TX, or VA.

TX 6-196-514

PA | PAU | SE | SEG | SEU | SR | SRU | TX | TXU | VA | VAU

EFFECTIVE DATE OF REGISTRATION

04 (Month) 20 (Day) 2005 (Year)

CONTINUATION SHEET RECEIVED
APR 20 2005

Page 3 of 4 pages

DO NOT WRITE ABOVE THIS LINE FOR COPYRIGHT OFFICE USE ONLY

A Identification of Application

IDENTIFICATION OF CONTINUATION SHEET This sheet is a continuation of the application for copyright registration on the basic form submitted for the following work.

- TITLE (Give the title as given under the heading "Title of this Work in Space 1 of the basic form)

Java 2 Standard Edition 1.4

- NAME(S) AND ADDRESS(ES) OF COPYRIGHT CLAIMANT(S) (Give the name and address of at least one copyright claimant as given in Space 4 of the basic form or Space 2 of any of the Short Forms PA, TX, or VA)

Sun Microsystems, Inc , 4150 Network Circle, Santa Clara, CA 95054

B Continuation of Space 2

d

NAME OF AUTHOR ▼: (SEE SPACE C)

DATES OF BIRTH AND DEATH
Year Born▼ Year Died▼

Was this contribution to the work a "work made for hire"?
☐ Yes ☐ No

AUTHOR S NATIONALITY OR DOMICILE
Name of Country
OR { Citizen of ▶ _____ / Domiciled in ▶ _____

WAS THIS AUTHOR S CONTRIBUTION TO THE WORK
Anonymous? ☐ Yes ☐ No
Pseudonymous? ☐ Yes ☐ No
If the answer to either of these questions is "Yes, see detailed instructions.

NATURE OF AUTHORSHIP Briefly describe nature of the material created by the author in which copyright is claimed ▼

e

NAME OF AUTHOR ▼

DATES OF BIRTH AND DEATH
Year Born▼ Year Died▼

Was this contribution to the work a "work made for hire"?
☐ Yes ☐ No

AUTHOR S NATIONALITY OR DOMICILE
Name of Country
OR { Citizen of ▶ _____ / Domiciled in ▶ _____

WAS THIS AUTHOR S CONTRIBUTION TO THE WORK
Anonymous? ☐ Yes ☐ No
Pseudonymous? ☐ Yes ☐ No
If the answer to either of these questions is "Yes, see detailed instructions.

NATURE OF AUTHORSHIP Briefly describe nature of the material created by the author in which copyright is claimed ▼

f

NAME OF AUTHOR ▼

DATES OF BIRTH AND DEATH
Year Born▼ Year Died▼

Was this contribution to the work a "work made for hire"?
☐ Yes ☐ No

AUTHOR S NATIONALITY OR DOMICILE
Name of Country
OR { Citizen of ▶ _____ / Domiciled in ▶ _____

WAS THIS AUTHOR S CONTRIBUTION TO THE WORK
Anonymous? ☐ Yes ☐ No
Pseudonymous? ☐ Yes ☐ No
If the answer to either of these questions is "Yes see detailed instructions.

NATURE OF AUTHORSHIP Briefly describe nature of the material created by the author in which copyright is claimed. ▼

Use the reverse side of this sheet if you need more space for continuation of Spaces 1 4 or 6 of the basic form or for the

CONTINUATION OF (Check which) ☐ Space 1 ☐ Space 4 ☐ Space 6 ☑ Space 2b

C
Continuation of other Spaces

Name of Author	Work for Hire	Domicile	Anonymous	Pseudo-nymous	Nature of Contribution
CNS Publishing Inc	Yes	United States	Yes	No	Text of Documentation Illustrations
Mary Dageforde dba Dageforde Consulting	Yes	United States	Yes	No	Text of Documentation
Chet Haase	No	United States	Yes	No	Computer code
PrO Unlimited, Inc	Yes	United States	Yes	No	Computer code Text of Documentation
Select Appointments (Holdings) PLC dba New Boston Systems Accountant s Inc AccountPros	Yes	United States	Yes	No	Computer code
Warewolf Technologies Inc	Yes	United States	Yes	No	Computer code
ZAO Elbrus MCST	Yes	Russia	Yes	No	Computer code

☑ **Space 5**

Previous Registration No	Year of Registration
TX 5 271 787	2000
TX 5 316 757	2000
TX 5 316-758	2000
TX 5-359 984	2001
TX 5 359 985	2001
TX 5 359 986	2001
TX 5 359 987	2001
TX 5 392 885	2001

Certificate will be mailed in window envelope to this address

Name ▼
Susanne S Morales, Paralegal / Fenwick & West LLP

Number/Street/Apt ▼
801 California Street

City/State/ZIP ▼
Mountain View, CA 94041

Complete all necessary spaces
Sign your application

D

1 Application form
2 Nonrefundable fee in check or money order payable to Register of Copyrights
3 Deposit Material

Library of Congress Copyright Office
101 Independence Avenue S E
Washington D C 20559-6000

Fees are subject to change. For current fees, check the Copyright Office website at www.copyright.gov, write the Copyright Office or call (202) 707-3000

Certificate of Registration

This Certificate issued under the seal of the Copyright Office in accordance with title 17, United States Code, attests that registration has been made for the work identified below. The information on this certificate has been made a part of the Copyright Office records.

Marybeth Peters

Register of Copyrights, United States of America

Form TX
For a Nondramatic Literary Work
UNITED STATES COPYRIGHT OFFICE

RE TX 6-066-538

EFFECTIVE DATE OF REGISTRATION
12 20 2004
Month Day Year

DO NOT WRITE ABOVE THIS LINE IF YOU NEED MORE SPACE, USE A SEPARATE CONTINUATION SHEET

1 TITLE OF THIS WORK ▼
Java 2 Standard Edition, Version 5.0

PREVIOUS OR ALTERNATIVE TITLES ▼
J2SE 5 0, Java 2 Platform, Standard Edition, Version 5.0

PUBLICATION AS A CONTRIBUTION If this work was published as a contribution to a periodical serial or collection give information about the collective work in which the contribution appeared Title of Collective Work ▼

If published in a periodical or serial give Volume ▼ Number ▼ Issue Date ▼ On Pages ▼

2 a NAME OF AUTHOR ▼
Sun Microsystems, Inc

DATES OF BIRTH AND DEATH
Year Born ▼ Year Died ▼

Was this contribution to the work a work made for hire ?
☑ Yes ☐ No

AUTHOR'S NATIONALITY OR DOMICILE
Name of Country
OR { Citizen of ▶ ____ Domiciled in ▶ United States

WAS THIS AUTHOR'S CONTRIBUTION TO THE WORK
Anonymous? ☐ Yes ☑ No
Pseudonymous? ☐ Yes ☑ No
If the answer to either of these questions is Yes see detailed instructions

NATURE OF AUTHORSHIP Briefly describe nature of material created by this author in which copyright is claimed ▼
New and revised computer code and accompanying documentation and manuals

NOTE
Under the law the author of a work made for hire is generally the employer not the employee (see instructions) For any part of this work that was made for hire check Yes in the space provided give the employer (or other person for whom the work was prepared) as Author of that part and leave the space for dates of birth and death blank

b NAME OF AUTHOR ▼
(SEE FORM TX/CON FOR ADDITIONAL AUTHORS)

DATES OF BIRTH AND DEATH
Year Born ▼ Year Died ▼

Was this contribution to the work a work made for hire ?
☐ Yes ☐ No

AUTHOR'S NATIONALITY OR DOMICILE
Name of Country
OR { Citizen of ▶ ____ Domiciled in ▶ ____

WAS THIS AUTHOR'S CONTRIBUTION TO THE WORK
Anonymous? ☐ Yes ☐ No
Pseudonymous? ☐ Yes ☐ No
If the answer to either of these questions is Yes see detailed instructions

NATURE OF AUTHORSHIP Briefly describe nature of material created by this author in which copyright is claimed ▼

c NAME OF AUTHOR ▼

DATES OF BIRTH AND DEATH
Year Born ▼ Year Died ▼

Was this contribution to the work a work made for hire ?
☐ Yes ☐ No

AUTHOR'S NATIONALITY OR DOMICILE
Name of Country
OR { Citizen of ▶ ____ Domiciled in ▶ ____

WAS THIS AUTHOR'S CONTRIBUTION TO THE WORK
Anonymous? ☐ Yes ☐ No
Pseudonymous? ☐ Yes ☐ No
If the answer to either of these questions is Yes see detailed instructions

NATURE OF AUTHORSHIP Briefly describe nature of material created by this author in which copyright is claimed ▼

3 a YEAR IN WHICH CREATION OF THIS WORK WAS COMPLETED This information must be given in all cases
2004 ◀ Year

b DATE AND NATION OF FIRST PUBLICATION OF THIS PARTICULAR WORK
Complete this information ONLY if this work has been published
Month ▶ September Day ▶ 30 Year ▶ 2004
United States ◀ Nation

4 COPYRIGHT CLAIMANT(S) Name and address must be given even if the claimant is the same as the author given in space 2 ▼
Sun Microsystems, Inc
4150 Network Circle
Santa Clara, CA 95054

See instructions before completing this space

TRANSFER If the claimant(s) named here in space 4 is (are) different from the author(s) named in space 2 give a brief statement of how the claimant(s) obtained ownership of the copyright ▼
By written agreement

DO NOT WRITE HERE OFFICE USE ONLY

APPLICATION RECEIVED
DEC 20 2004

ONE DEPOSIT RECEIVED
DEC 20 2004

TWO DEPOSITS RECEIVED

FUNDS RECEIVED

MORE ON BACK ▶ Complete all applicable spaces (numbers 5 9) on the reverse side of this page
See detailed instructions Sign the form at line 8

DO NOT WRITE HERE
Page 1 of 4

EXAMINED BY

CHECKED BY

FORM TX

☐ CORRESPONDENCE Yes

FOR COPYRIGHT OFFICE USE ONLY

DO NOT WRITE ABOVE THIS LINE IF YOU NEED MORE SPACE, USE A SEPARATE CONTINUATION SHEET

PREVIOUS REGISTRATION Has registration for this work or for an earlier version of this work already been made in the Copyright Office?

☑ Yes ☐ No If your answer is Yes why is another registration being sought? (Check appropriate box) ▼

a ☐ This is the first published edition of a work previously registered in unpublished form

b ☐ This is the first application submitted by this author as copyright claimant

c ☑ This is a changed version of the work as shown by space 6 on this application

If your answer is Yes give **Previous Registration Number** ▶

(SEE FORM TX/CON FOR PREVIOUS REGISTRATIONS)

Year of Registration ▶

5

DERIVATIVE WORK OR COMPILATION

Preexisting Material Identify any preexisting work or works that this work is based on or incorporates ▼

a

Prior works by claimant and licensed-in components

6

See instructions before completing this space

Material Added to This Work Give a brief general statement of the material that has been added to this work and in which copyright is claimed ▼

b

New and revised computer code and accompanying documentation and manuals

DEPOSIT ACCOUNT If the registration fee is to be charged to a Deposit Account established in the Copyright Office give name and number of Account

Name ▼

Account Number ▼

a 7

CORRESPONDENCE Give name and address to which correspondence about this application should be sent Name/Address/Apt/City/State/ZIP ▼

b

Ines Gonzalez, Esq
Fenwick & West LLP
801 California Street
Mountain View, CA 94041

Area code and daytime telephone number ▶ (650) 335-7182

Fax number ▶ (650) 938-5200

Email ▶ igonzalez@fenwick com

CERTIFICATION* I the undersigned hereby certify that I am the

Check only one ▶

- ☐ author
- ☐ other copyright claimant
- ☐ owner of exclusive right(s)
- ☑ authorized agent of Sun Microsystems, Inc

Name of author or other copyright claimant or owner of exclusive right(s) ▲

of the work identified in this application and that the statements made by me in this application are correct to the best of my knowledge

8

Typed or printed name and date ▼ If this application gives a date of publication in space 3 do not sign and submit it before that date

Marilyn E Glaubensklee, Assistant General Counsel

Date ▶ 12/17/04

Handwritten signature (X) ▼

X Marilyn E Glaubensklee

Certificate will be mailed in window envelope to this address

Name ▼
Susanne S Morales, Paralegal / Fenwick & West LLP

Number/Street/Apt ▼
801 California Street

City/State/ZIP ▼
Mountain View, CA 94041

YOU MUST:
Complete all necessary spaces
Sign your application in space 8

SEND ALL 3 ELEMENTS IN THE SAME PACKAGE:
1 Application form
2 Nonrefundable filing fee in check or money order payable to *Register of Copyrights*
3 Deposit material

MAIL TO:
Library of Congress
Copyright Office TX
101 Independence Avenue S E
Washington D C 20559 6222

9

Fees are subject to change For current fees check the Copyright Office website at www copyright gov write the Copyright Office or call (202) 707-3000

17 U S C § 506(e) Any person who knowingly makes a false representation of a material fact in the application for copyright registration provided for by section 409 or in any written statement filed in connection with the application shall be fined not more than $2 500

CONTINUATION SHEET FOR APPLICATION FORMS

Form TX /CON
UNITED STATES COPYRIGHT OFFICE

R TX 6-066-538

- This Continuation Sheet is used in conjunction with Forms CA PA SE SR TX and VA only Indicate which basic form you are continuing in the space in the upper right hand corner
- If at all possible try to fit the information called for into the spaces provided on the basic form
- If you do not have enough space for all the information you need to give on the basic form use this Continuation Sheet and submit it with the basic form
- If you submit this Continuation Sheet clip (do not tape or staple) it to the basic form and fold the two together before submitting them
- **Space A of this sheet is intended to identify the basic application**
  Space B is a continuation of Space 2 on the basic application
  Space B is not applicable to Short Forms
  Space C (on the reverse side of this sheet) is for the continuation of Spaces 1, 4, or 6 on the basic application or for the continuation of Space 1 on any of the three Short Forms PA, TX, or VA

PA	PAU	SE	SEG	SEU	SR	SRU	TX	TXU	VA	VAU

EFFECTIVE DATE OF REGISTRATION

12 (Month) 20 (Day) 2004 (Year)

CONTINUATION SHEET RECEIVED
DEC 20 2004

Page 3 of 4 pages

DO NOT WRITE ABOVE THIS LINE FOR COPYRIGHT OFFICE USE ONLY

A Identification of Application

IDENTIFICATION OF CONTINUATION SHEET This sheet is a continuation of the application for copyright registration on the basic form submitted for the following work

- TITLE (Give the title as given under the heading Title of this Work in Space 1 of the basic form)

Java 2 Standard Edition, Version 5 0

- NAME(S) AND ADDRESS(ES) OF COPYRIGHT CLAIMANT(S) (Give the name and address of at least one copyright claimant as given in Space 4 of the basic form or Space 2 of any of the Short Forms PA TX or VA)

Sun Microsystems, Inc , 4150 Network Circle, Santa Clara, CA 95054

B Continuation of Space 2

d

NAME OF AUTHOR ▼ (SEE SPACE C)

DATES OF BIRTH AND DEATH
Year Born▼ Year Died▼

Was this contribution to the work a work made for hire ?
☐ Yes ☐ No

AUTHOR S NATIONALITY OR DOMICILE
Name of Country
OR { Citizen of ▶ _____ / Domiciled in ▶ _____

WAS THIS AUTHOR S CONTRIBUTION TO THE WORK
Anonymous? ☐ Yes ☐ No
Pseudonymous? ☐ Yes ☐ No
If the answer to either of these questions is Yes see detailed instructions

NATURE OF AUTHORSHIP Briefly describe nature of the material created by the author in which copyright is claimed ▼

e

NAME OF AUTHOR ▼

DATES OF BIRTH AND DEATH
Year Born▼ Year Died▼

Was this contribution to the work a work made for hire ?
☐ Yes ☐ No

AUTHOR'S NATIONALITY OR DOMICILE
Name of Country
OR { Citizen of ▶ _____ / Domiciled in ▶ _____

WAS THIS AUTHOR S CONTRIBUTION TO THE WORK
Anonymous? ☐ Yes ☐ No
Pseudonymous? ☐ Yes ☐ No
If the answer to either of these questions is Yes see detailed instructions

NATURE OF AUTHORSHIP Briefly describe nature of the material created by the author in which copyright is claimed ▼

f

NAME OF AUTHOR ▼

DATES OF BIRTH AND DEATH
Year Born▼ Year Died▼

Was this contribution to the work a work made for hire ?
☐ Yes ☐ No

AUTHOR'S NATIONALITY OR DOMICILE
Name of Country
OR { Citizen of ▶ _____ / Domiciled in ▶ _____

WAS THIS AUTHOR S CONTRIBUTION TO THE WORK
Anonymous? ☐ Yes ☐ No
Pseudonymous? ☐ Yes ☐ No
If the answer to either of these questions is Yes see detailed instructions

NATURE OF AUTHORSHIP Briefly describe nature of the material created by the author in which copyright is claimed ▼

Use the reverse side of this sheet if you need more space for continuation of Spaces 1 4 or 6 of the basic form or for the

CONTINUATION OF (Check which) ☐ Space 1 ☐ Space 4 ☐ Space 6 ☑ Space 2b

C
Continuation of other Spaces

Name of Author	Work for Hire	Domicile	Anonymous	Pseudonymous	Nature of Contribution
Comsys	Yes	United States	Yes	Yes	Computer code and Text of documentation
PrOUnlimited, Inc	Yes	United States	Yes	Yes	Text of documentation
TelTech International Corp	Yes	United States	Yes	Yes	Computer code
The Carl Group	Yes	United States	Yes	Yes	Text of documentation
ZAO Elbrus MCST	Yes	United States	Yes	Yes	Computer code and Text of documentation

☑ Space 5

Previous Registration No	Year of Registration
TX 5-271-787	2000
TX 5-316-757	2000
TX 5-316-758	2000
TX 5-359-984	2001
TX 5-359-985	2001
TX 5-359-986	2001
TX 5-359-987	2001
TX 5-392-885	2000

Certificate will be mailed in window envelope to this address

Name ▼
Susanne S Morales, Paralegal / Fenwick & West LLP

Number/Street/Apt ▼
801 California Street

City/State/ZIP ▼
Mountain View, CA 94041

YOU MUST:
Complete all necessary spaces
Sign your application

SEND ALL 3 ELEMENTS IN THE SAME PACKAGE:
1 Application form
2 Nonrefundable fee in check or money order payable to Register of Copyrights
3 Deposit Material

MAIL TO:
Library of Congress Copyright Office
101 Independence Avenue S E
Washington D C 20559 6000

D

Fees are subject to change For current fees check the Copyright Office website at www.copyright.gov write the Copyright Office or call (202) 707 3000

Certificate of Registration

Form CA
For Supplementary Registration
UNITED STATES COPYRIGHT OFFICE

REGISTRATION NUMBER
TX 6-143-306

TX | TXU | PA | PAU | VA | VAU | SR | SRU | RE

EFFECTIVE DATE OF SUPPLEMENTARY REGISTRATION
FEB 2 2005
Month Day Year

This Certificate issued under the seal of the Copyright Office in accordance with title 17, United States Code, attests that registration has been made for the work identified below. The information on this certificate has been made a part of the Copyright Office records.

Marybeth Peters
Register of Copyrights, United States of America

DO NOT WRITE ABOVE THIS LINE. IF YOU NEED MORE SPACE, USE A SEPARATE CONTINUATION SHEET.

Title of Work ▼
Java 2 Standard Edition, Version 5.0

Registration Number of the Basic Registration ▼	**Year of Basic Registration ▼**
TX 6-066-538	2004

Name(s) of Author(s) ▼	**Name(s) of Copyright Claimant(s) ▼**
Sun Microsystems, Inc. Comsys — The Carl Group PrO Unlimited — ZAO Elbrus MCST TelTech International Corp.	Sun Microsystems, Inc.

Location and Nature of Incorrect Information in Basic Registration ▼

Line Number ____ Line Heading or Description ____

Incorrect Information as It Appears in Basic Registration ▼

Corrected Information ▼

Explanation of Correction ▼

Location and Nature of Information in Basic Registration to be Amplified ▼

Line Number 1 Line Heading or Description Title Of The Work

Amplified Information and Explanation of Information ▼

The following titles were inadvertently omitted from the basic registration and should be added as alternative titles of the work:

Java 2 Standard Edition 5.0 Development Kit
Java 2 Platform Standard Edition 5.0 Development Kit
J2SE Development Kit
JDK 5.0

MORE ON BACK ▶ • Complete all applicable spaces (D-G) on the reverse side of this page. • See detailed instructions. • Sign the form at Space F.

DO NOT WRITE HERE
Page 1 of 2 pages.

FORM CA

FORM CA RECEIVED

FEB. 0 2 2005

FUNDS RECEIVED DATE

EXAMINED BY

CORRESPONDENCE ☐

REFERENCE TO THIS REGISTRATION ADDED TO BASIC REGISTRATION ☐ YES ☐ NO

FOR COPYRIGHT OFFICE USE ONLY

DO NOT WRITE ABOVE THIS LINE. IF YOU NEED MORE SPACE, USE A SEPARATE CONTINUATION SHEET.

Continuation of: ☐ Part B *or* ☐ Part C

Correspondence: Give name and address to which correspondence about this application should be sent.

Ines Gonzalez, Esq.
Fenwick & West LLP
801 California Street
Mountain View, CA 94041

Phone (650) 335-7182 Fax (650) 938-5200 Email igonzalez@fenwick.com

Deposit Account: If the registration fee is to be charged to a Deposit Account established in the Copyright Office, give name and number of Account.

Name

Account Number

Certification* I, the undersigned, hereby certify that I am the: (Check only one)

☐ author ☐ owner of exclusive right(s)
☐ other copyright claimant ☑ duly authorized agent of Sun Microsystems, Inc.
Name of author or other copyright claimant, or owner of exclusive right(s) ▲

of the work identified in this application and that the statements made by me in this application are correct to the best of my knowledge.

Typed or printed name ▼ Marilyn E. Glaubensklee, Assistant General Counsel **Date ▼** 1/31/05

Handwritten signature (X) ▼ Marilyn E. Glaubensklee

Certificate will be mailed in window envelope to this address:

Name ▼
Susanne S. Morales, Paralegal / Fenwick & West LLP

Number/Street/Apt ▼
801 California Street

City/State/ZIP ▼
Mountain View, CA 94041

YOU MUST:
- Complete all necessary spaces
- Sign your application in Space F

SEND ALL ELEMENTS IN THE SAME PACKAGE:
1. Application form
2. Nonrefundable filing fee in check or money order payable to *Register of Copyrights*

MAIL TO:
Library of Congress
Copyright Office
101 Independence Avenue, S.E.
Washington, D.C. 20559-6000

Fees are subject to change. For current fees, check the Copyright Office website at www.copyright.gov, write the Copyright Office, or call (202) 707-3000.

*17 U.S.C. § 506(e): Any person who knowingly makes a false representation of a material fact in the application for copyright registration provided for by section 409, or in any written statement filed in connection with the application, shall be fined not more than $2,500.